# Reliable Cloud Services with Byzantine Fault Tolerance

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades einer

**Doktoringenieurin (Dr.-Ing.)**

genehmigte Dissertation

von

Bijun Li
geboren am 04.08.1986
in Shandong, China

| | |
|---|---|
| Eingereicht am: | 10.06.2021 |
| Disputation am: | 29.09.2021 |
| 1. Referent: | Prof. Dr. Rüdiger Kapitza |
| 2. Referent: | Prof. Dr. Hans Reiser |

2021

# Abstract

Cloud computing, a computer technology based on the Internet, has developed rapidly in the last decade. Nowadays, cloud vendors offer different types of cloud computing products to customers. Compared to local servers in the data centers, the use of clouds can bring significant benefits to customers, such as flexible scalability, high availability in terms of running stateless services, and reduced infrastructure maintenance costs, to name a few. For this reason, there has been a massive upward trend in cloud computing in recent years, with more and more Internet service providers preferring to run their services, such as social networks and online streaming, on cloud infrastructure rather than on local servers.

Apart from the above-mentioned Internet services used in our daily lives, there is also such a trend in traditional infrastructure-related services. Services such as traffic control and railway network management are gradually being integrated into cloud infrastructures. Unlike those commodity services, infrastructure services can be critical as they usually have an incredible responsibility to society, which means that any failure events can lead to fatal consequences. However, since most existing cloud infrastructures do not guarantee fault tolerance for dependable stateful services, which mostly consist of infrastructure services, they need to be strengthened to provide higher reliability and availability to these services.

In this thesis, we present the design for building a cloud environment that provides reliable services with Byzantine Fault Tolerance (BFT) that is resilient not only to crashes but also to arbitrary failures of system components. In order to make traditional cloud services to tolerate Byzantine failures and guarantee good performance, this thesis presents a set of solutions that address the problems from different angles.

First, we introduce a parallel ordering framework that relies on multiple leaders to improve the performance and scalability of BFT systems. Under this framework, multiple BFT protocol instances run within each replica for concurrent ordering and execution of independent requests. Second, we present a prototype that leverages trusted execution environments to provide transparent access to BFT systems. It implements a replacement for the library required for client access within the trusted environment to allow legacy clients to access the replicated services without modification. We also introduce a novel read optimization to further accelerate the processing of read-heavy workloads while providing strong consistency guarantees. Third, we implement a framework that can automate the deployment and evaluation of BFT systems with different software configurations. The use of this framework can avoid the error-prone manual management and orchestration of replicated services in the cloud environment and greatly facilitate the comparison of different fault-tolerant systems.

# Kurzzusammenfassung

Cloud Computing hat sich als internetbasierte Technologie innerhalb des letzten Jahrzehnts rasant weiterentwickelt. Heutzutage bieten Cloud Anbieter den Kunden die unterschiedlichsten Arten von Cloud Computing Produkten an. Im Vergleich zu lokalen Servern kann die Nutzung von Clouds den Kunden signifikante Vorteile, wie beispielsweise flexible Skalierbarkeit und Hochverfügbarkeit beim Betrieb von zustandslosen Diensten, sowie eine Senkung der Wartungskosten für die Infrastruktur, bieten. Aus diesen Gründen lässt sich in den letzten Jahren ein Aufwärtstrend beim Wachstum von Cloud Computing beobachten, so dass immer mehr Internet Service Provider es bevorzugen ihre Dienste, wie beispielsweise soziale Netzwerke und Online-Streaming in einer Cloud Infrastruktur anstatt auf lokalen Server zu betreiben.

Neben den zuvor erwähnten Internetdiensten, die wir im täglichen Leben benutzen, gibt es auch einen Trend, dass herkömmliche Dienste, die sich auf Infrastrukturanlagen, wie beispielsweise die Verkehrsregelung im Straßenverkehr und die Verwaltung eines Schienenverkehrsnetzes beziehen, schrittweise in Cloud Infrastrukturen überführt werden. Im Gegensatz zu anderen kommerziellen Internetdiensten gelten diese Dienste als kritisch, da jedes Auftreten eines Fehlers zu verhängnisvollen Konsequenzen führen und Menschenleben gefährden könnte. Jedoch bieten die meisten bestehenden Cloud Infrastrukturen keine ausreichende Unterstützung für Fehlertoleranz von zustandsorientierten Diensten in diesem kritischen Umfeld. Daher muss bei diesen eine höhere Zuverlässigkeit und Verfügbarkeit sichergestellt werden können.

In dieser Dissertation wird der Entwurf von zuverlässigen Diensten unter Anwendung byzantinischer Fehlertoleranz (BFT) in einer Cloud Umgebung vorgestellt. Die so abgesicherten Dienste sind widerstandsfähig gegen Abstürze sowie willkürliche Fehler einzelner Systemkomponenten. Um bei traditionellen Cloud Diensten byzantinische Fehler zu tolerieren und zudem eine gute Leistungsfähigkeit zu garantieren, zeigt diese Dissertation eine Reihe von Lösungen auf, um diese Probleme zu bewältigen.

Erstens, wird ein Parallel Ordering Framework, das sich auf mehrere Anführer stützt vorgestellt, um die Leistungsfähigkeit und die Skalierbarkeit von Systemen mit byzantinischer Fehlertoleranz zu verbessern. In diesem Framework laufen mehrere BFT-Instanzen in jedem Replica für nebenläufiges Ordering und Ausführung von unabhängigen Anfragen. Zweitens, wird ein Prototyp vorgestellt, um bestehenden Clients Zugriff ohne Anpassungen auf die replizierten Dienste zu ermöglichen. Dieser Prototyp ermöglicht die Ausführung in einer vertrauenswürdigen Laufzeitumgebung, um einen transparenten Zugriff auf BFT-Systeme zu ermöglichen. Darüber hinaus wird eine neuartige Optimierung zur Beschleunigung von Lesevorgängen eingeführt, während weiterhin eine hohe Konsistenz garantiert wird. Drittens, wird durch die Implementierung eines Frameworks aufgezeigt, wie die Bereitstellung und die Evaluierung von BFT-Systemen mit verschiedenen Softwarekonfigurationen automatisierbar ist. Durch die Nutzung des Frameworks wird die fehleranfällige,

manuelle Verwaltung und Orchestrierung von replizierten Diensten in der Cloud Infrastruktur vermieden und zusätzlich der Vergleich von unterschiedlichen fehlertoleranten Systemen ermöglicht.

# Acknowledgement

First and foremost, I am extremely grateful to my doctoral supervisor, Prof. Rüdiger Kapitza, for his guidance and continuous support during my doctoral studies. His immense expertise and experience encouraged me to overcome all obstacles during this research. I would like to thank Prof. Jens Braband for giving me the opportunity to be a part of Siemens Rail Automation Graduate School. Thanks to Prof. Lars Wolf for making IBR a home for us.

I would also like to express my gratitude and appreciation to the following people, without whom I could not have completed this research: Johannes Behl, Tobias Distler, Pierre-Louis Aublin, Wenbo Xu, Nico Weichbrodt and Muhammad Zeeshan Abid, for their significant support and contribution to the research topics. Arthur Martens, Björn Cassens, Vasily A. Sartakov, Stefan Brenner, Marcus Brandenburger, David Goltzsche, Signe Rüsch, Manuel Nieke and Ines Messadi, all members of the IBR DS group, for being so kind and supportive during these years and making DS a place with fond memories. Yinghui, Imko, Antje and Mareike who helped me out with everything when I first came to Germany.

Finally, I would like to thank my parents, in-laws and other family members for supporting me until today, for all the encouragement they have given me during these years.

Last but not least, I thank my beloved husband Hendrik for all the love and support, without which I would have stopped this journey a long time ago.

# Contents

# List of Figures

# List of Tables

# 1
# Introduction

## 1.1 Motivation

Cloud computing [1, 2] has become a massive trend in the last decade as it offers a new form of resource provisioning. In cloud computing, system resources and application-level services are no longer hosted on local machines but are provided by a shared remote infrastructure owned by cloud providers. By leveraging cloud infrastructure resources delivered over the Internet, customers can achieve significant benefits, such as flexible scalability of resource usage, low cost, and low complexity for management by avoiding the underlying infrastructure management tasks.

Different types of clouds are capable of providing cloud infrastructure resources at different levels of abstraction, including virtual machines, software deployment platforms with operating systems, runtime environments, and application software. More and more Internet services, such as cloud storage and online streaming, rely on cloud infrastructure resources instead of local data centers to provide their services. However, due to economic pressures, the underlying cloud infrastructure is usually built from hundreds of thousands of commodity machines networked together via switches and routers, which means that the deployed hardware is exposed to scale and conditions for which it was not designed [3]. Because of this, failures of individual components within the cloud infrastructure can occur frequently. For example, a new cluster from Google may have experienced thousands of hardware failures in its first year caused by various components [4]. For the Internet services deployed on the cloud infrastructure, their availability depends heavily on the reliability of the underlying cloud and is therefore immediately affected when a cloud infrastructure failure occurs, as shown in Figure 1.1. Examples of such cases can often be found in the news as the outages of well-known Internet services are caused by the underlying cloud outages [5, 6, 7]. Therefore, to ensure high reliability of the deployed services, it is imperative to have fault tolerance software for the cloud infrastructure.

Most existing cloud infrastructures solve this problem by applying a non-functional property called horizontal scalability, which replicates a service and keeps the replicas running at the same time. Together with a load balancer, namely the High Availability Proxy (HAProxy) [8], horizontal scalability is able to tolerate service crashes and handle load balancing. However, this can only be applied to stateless services, as the processing of distributed workloads may result in different service states across replicas. Certain commodity Internet services can benefit from using horizontal scalability as long as their outputs depend solely on the value of the inputs.

In recent years, with the rapid development and popularization of cloud technology many legacy services, e.g., services related to social infrastructures such as traffic control and railway network management, have been gradually integrated into the cloud infrastructure. On the one hand, social services are quite critical as their failures could lead to fatal consequences for society, so they should be resilient not only against crashes [9] but also against malicious behavior of system components [10, 11, 12]. On the other hand, the integration of such services imposes additional requirements on the cloud infrastructure that cannot be easily met under current conditions. Since the availability of the deployed services is highly dependent on the reliability of the underlying cloud infrastructure, it is essential for cloud

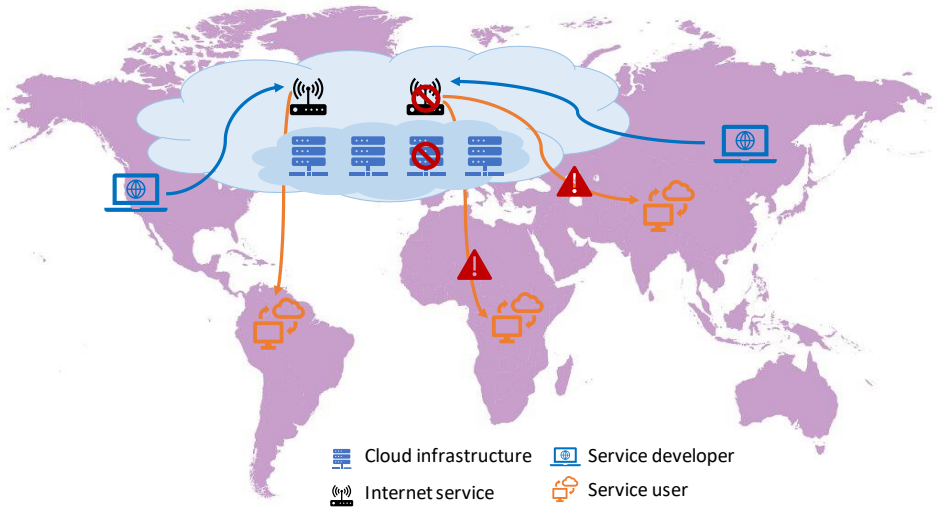



Figure 1.1: The availability of Internet services depends on the reliability of the underlying cloud infrastructures.

providers to find out a solution for stateful service replication in the cloud.

In this thesis, we assume that client applications and services (hereafter referred to as services) deployed in a cloud environment should tolerate not only crash-stop failures, but also arbitrary and malicious behavior, also known as Byzantine failures. To achieve this goal, we build a system with *Byzantine fault tolerance* (BFT) [13] to ensure that the reliability of replicated stateful services can be guaranteed by achieving consensus across all service replicas. With this guarantee, the deployed services can tolerate both crashes and software errors, e.g., arbitrary database behavior, to avoid service outage incidents. Moreover, it is also resilient to failures caused by malicious and intentionally colluding adversaries that try to prevent correct replicas from delivering their services, e.g., via intrusions [14, 15, 16, 17, 18].

Although Byzantine fault-tolerant systems can do a good job protecting replicated services by improving the reliability and availability of those services, they have not yet been widely used in production environments [19]. The reason for this may be multifaceted:

- Byzantine fault-tolerant systems typically have high resource demand and introduce additional overhead to services, which can easily lead to performance problems. In a classic scenario of a Byzantine fault-tolerant system, at least $3f + 1$ replicas are required to tolerate $f$ failures, which significantly limits scalability and makes the system resource-intensive. Meanwhile, all replicas must participate in the execution of a complex agreement protocol to order each request. The agreement protocol typically involves three rounds of all-to-all message exchange, which causes additional resource consumption and overhead for the replicas. In the last two decades, many

excellent research works have been proposed to solve these problems in various ways, e.g., by improving performance [20, 21, 22, 23, 24, 25, 26, 27], increasing scalability [28, 29, 30, 31], reducing implementation costs [32, 33, 34, 35], and increasing resilience [36, 37, 38, 39]. However, even with the performance improvements achieved by these approaches, the concurrency problem in Byzantine fault-tolerant systems remains unsolved, as they mostly rely on a total order of all requests to maintain consistency across replicas. This leads to large overhead and limits the system performance.

- Another problem is found in classical Byzantine fault-tolerant systems, and it inherently prevents them from being widely deployed in popular cloud environments. In order to access replicated services, clients must integrate a client-side library of the BFT protocol that includes essential functions such as communicating with replicas and verifying reply messages. However, integrating such a library into the client implementations can hardly be practical and is basically contradictory to the use of clouds and Internet services. This is because typical Internet services are mostly user-facing, and should be provided through a universal interface without requiring any changes on the client side. Therefore, the client-side BFT library should be removed while its functionality must be preserved.

- Also, deploying and configuring the replicated stateful services in the cloud environment is cumbersome as the existing cloud environments usually do not support such requirements. In this case, the replication and orchestration processes need to be performed manually by the customers, which can easily lead to complicated and error-prone tasks for management. Therefore, an approach that enables automated deployment and evaluation of Byzantine fault-tolerant systems and reduces the difficulties of running replicated stateful services in the cloud is absolutely needed.

## 1.2 Purpose of This Thesis

In this thesis, we propose a design that improves the performance, reliability and usability of Byzantine fault-tolerant systems, especially in cloud environments. The implementation of this design involves a series of solutions that are carried out in three steps. First, a performance problem of Byzantine fault-tolerant systems is stated and solved by using parallelism. Second, it is shown that the client-side BFT library is not applicable to most clients of Internet services, and a novel approach is presented that can provide transparent access to the replicated services. Third, to avoid the complicated and error-prone manual management of the replicated services in the cloud, it provides support for handling the deployment and evaluation of Byzantine fault-tolerant systems with automation tools.

### 1.2.1 Performance Improvement

Along with hardware development, such as multi-core technology, state machine replication has been given the ability to exploit the potential to process non-conflicting requests in par-

allel with multiple cores [40, 41, 42, 43, 44]. Parallelization offers great advantages for use cases where the actual request processing time is not negligible, and has been applied to many Byzantine fault-tolerant systems in the execution stage.

However, parallelization has not reached the agreement stage yet, so existing Byzantine fault-tolerant systems still rely on a single leader at a time for the ordering of all requests. To enable parallelism in both the agreement and execution stages, in this thesis we present a framework that executes multiple Byzantine fault-tolerant protocol instances and uses multiple leaders concurrently. The concurrent instances have access to the same service state, which in practice is divided into several disjoint partitions, and each instance is only responsible for maintaining a partial sequence of requests accessing one of the partitions. As a result, all instances can order and execute independent (i.e., without interfering updates) requests in parallel, ultimately improving performance. In addition, the state partitioning approach has been optimized to reduce the possibility of imperfect partitions due to inaccurate partitioning knowledge.

### 1.2.2 Transparent Access

Although modern Byzantine fault-tolerant systems have shown that they can achieve good performance sufficient for use in real production environment [27, 35], there is still a problem that is widely overlooked. Despite the good performance, it is not possible for clients of Byzantine fault-tolerant systems to access the replicated services transparently (i.e., without modification by the client-side BFT protocol library). The client-side library contains essential functionality for communication and reply voting and is therefore indispensable for the system to accomplish the request processing operation. However, adapting this library is hardly feasible for most user-facing services (e.g., Web service, e-mail service) because they already use many established protocols (e.g., HTTP, IMAP) that can hardly be modified, and the client implementations of these services differ significantly.

To obtain transparent access to the replicated services, the functionality of the client-side library must be moved to the server side. However, since Byzantine fault-tolerant systems assume a failure model where any replica can fail arbitrarily, there must be a solution for the security and integrity guarantee of the shifted functionality. To achieve this, we rely on a trusted subsystem that provides a trusted execution environment so that the shifted functionality can be executed within the environment in a protected manner. It also ensures that it is trusted even if the host replica becomes faulty.

### 1.2.3 Automated Deployment and Evaluation

Deploying replicated stateful services in the existing cloud environments imposes additional requirements on the management and coordination of replicated services across replicas that are not met by the existing clouds. Nevertheless, it requires a lot of effort for the customers to manage the manual deployment and orchestration of Byzantine fault-tolerant systems and the replicated services. Moreover, the configuration of the evaluation process also needs to be optimized, which can be error-prone and easily leads to misconfigurations.

Therefore, it is necessary to build a framework in the cloud environment that automates the entire process of deployment and evaluation, and ensures the framework can meet the resource utilization requirements, etc. of the various services. By using this framework, customers can easily deploy and run their services with specific Byzantine fault-tolerant systems to ensure reliability. Moreover, they can also perform the evaluation process and result analysis in an automated manner.

## 1.3 Scope of This Thesis

This thesis aims to improve the performance, reliability, and usability of client-provided services in a cloud environment by using Byzantine fault tolerance. To achieve this, we replicate the client-provided services in the cloud, where the cloud infrastructure has been enhanced with resilience against Byzantine faults. The replicated services should be distributed across multiple physical hosts of the cloud to reduce fault dependency, based on the assumption that faults in different host replicas are uncorrelated [12, 45, 46]. We also consider system diversity in terms of replica host configurations, since replicas with the same configuration (e.g., the same operating system) are vulnerable to certain errors or malicious attacks [47], especially when the same configuration vulnerability is explored and exploited by the attacker. Therefore, in this thesis, we succeed in building a system consisting of replicas with different configurations. However, it does not cover the possible solutions to reduce the causes of such flaws.

Regarding the errors caused by malicious attacks, this thesis assumes a trusted subsystem that is intrusion-tolerant to ensure that an essential function of the system is trusted even in the presence of attacks. However, other security-related topics are not addressed in this thesis, such as techniques like access control to protect against denial-of-service attacks [48, 37], or confidentiality protection [28, 49].

## 1.4 Structure of This Thesis

The remainder of this thesis is as follows:

**Chapter 2** presents the basics of this thesis, including background knowledge of Byzantine fault-tolerant systems and the system model of the protocols used in this thesis.

**Chapter 3** analyzes the issues in building a cloud environment for running fault-tolerant applications with high performance. In addition, this chapter reviews the proposed solutions to overcome the following challenges: (1) using a BFT protocol with parallel ordering and execution to improve performance, (2) moving the client-side library of the BFT protocol to the server side for transparent access, and (3) implementing a framework for automated deployment and evaluation of replicated services in the cloud environment.

**Chapter 4** first introduces SAREK, a multi-leader-based framework that partitions the entire service state into partitions to leverage parallelism in both the agreement and execution stages. It also presents DYPART, which abstracts the access patterns of each application to the state objects to generate application-specific request dependencies. Based on this, a deterministic partitioning algorithm can be executed to improve quality and reduce synchronization overhead across partitions.

**Chapter 5** introduces TROXY, a novel design that moves the functionality of the client-side BFT library to the server side, allowing legacy clients to access replicated services without modification. It is implemented by using a trusted subsystem that provides a trusted execution environment to guarantee the reliability and integrity of the relocated functionality.

**Chapter 6** presents a framework for automated deployment and evaluation of Byzantine fault-tolerant systems and replicated services in a cloud environment. This framework leverages existing cloud technologies and automation tools to solve specific problems such as replication and orchestration of stateful services and management of different software configurations.

**Chapter 7** concludes the contributions of this thesis and discusses directions for further research based on this thesis.

## 1.5 Related Publications

The approaches and results presented in this thesis have been published as:

[50] B. Li, W. Xu, M. Z. Abid, T. Distler, and R. Kapitza, "Sarek: Optimistic parallel ordering in byzantine fault tolerance," in *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 2016, pp. 77–88

[51] B. Li, W. Xu, and R. Kapitza, "Dynamic state partitioning in parallelized byzantine fault tolerance," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 158–163

[52] B. Li, N. Weichbrodt, J. Behl, P.-L. Aublin, T. Distler, and R. Kapitza, "Troxy: Transparent access to byzantine fault-tolerant systems," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 59–70

[53] B. Li and R. Kapitza, "Bft-dep: automatic deployment of byzantine fault-tolerant services in paas cloud," in *Distributed Applications and Interoperable Systems*. Springer, 2016, pp. 109–114

I was the lead author, principal designer and implementer of the systems in [51, 53]. In [50, 52] I was the lead author and one of the main contributors to the design and implementation of the systems.

# 2
# System Model and Background

In this chapter, the system model of Byzantine fault-tolerant systems relevant to this thesis and the background of cloud computing are introduced. First, the common assumptions in BFT systems and the classical BFT system model are explained. In addition, this chapter shows the architecture and functionality of different types of cloud computing models, as well as their limitations.

## 2.1 System Model and Architecture

In this section we present the system model of Byzantine fault-tolerant systems, along with different fault models, namely the classical Byzantine fault model and a hybrid fault model that uses a trusted execution environment for additional reliability. These models serve as the basis for the proposed approaches presented in the following chapters.

### 2.1.1 System Model

In this thesis, we consider a typical distributed system model, where an application runs in each *server* on the server side and the *clients* from the client side use the deployed services of the application. To provide the deployed application with fault tolerance, the system model introduces consensus based on *state machine replication*: Replicated servers, called *replicas*, run on individual physical machines and are initialized with identical states.

#### 2.1.1.1 Communication

The communication process is performed on both the client and server sides in the following manner: (1) A client can access the offered services by issuing requests to the replicas so that the latter invoke operations to process them. (2) The operations may cause state changes on each replica and thus must be invoked and executed in a predefined order, since executing the same set of operations in different orders will cause divergences in the state machines. (3) Replicas communicate with each other through the server-side network connections by exchanging messages to achieve deterministic order of operations and solve the consensus problem. (4) After a replica executes the operation invoked by a request, it returns the result to the client in a response.

The underlying network between the components of the system is not necessarily reliable, so the transmitted messages may be delayed, corrupted, transmitted in the wrong order, or even dropped due to network connectivity errors or malicious attacks. We assume that such problems, except for network partitions, can be handled by lower network layers (e.g., TCP layer). Messages are authenticated on both the client and server sides. We also assume that it is not possible to break the cryptographic techniques (e.g., symmetric cryptography or public-key signature). Therefore, an attacker is not able to send messages on behalf of a non-faulty client or replica without being detected. If the verification of a message fails, i.e., the actual sender is not the one specified in the message, the message is discarded before it is processed.

#### 2.1.1.2 Safety and Liveness

As described in the PBFT [48] approach, providing *safety* and *liveness* properties can ensure the correctness of a Byzantine fault-tolerant system, if no more than $f = \lfloor \frac{n-1}{3} \rfloor$ replicas out of a total number of $n$ replicas are faulty.

In this context, the safety property is defined as "the replicated service behaves like a centralized implementation to execute operations atomically one at a time" [48]. The safety

property can guarantee that in a Byzantine fault-tolerant system, the states of non-faulty replicas remain consistent as long as no more than $f$ replicas are faulty. In a static server cluster where replicas do not scale dynamically, the number of faulty replicas $f$ is also predefined and remains constant. In this case, this means that the moment the number of faulty replicas exceeds this threshold, all fault-tolerant guarantees are no-longer valid. The safety in the system ensures that a client receives a sufficient number of identical and correct results to its requests, which has the same effect as processing the requests through a centralized service implementation.

The liveness property means that "clients eventually receive replies to their requests" [48]. This property holds due to the fact that in a fully asynchronous system where the communication delay is unbounded, it is impossible to reach consensus even if a single node crashes (FLP impossibility [54]), which means that liveness is only valid if a Byzantine fault-tolerant system assumes a *partially synchronous* environment [55]. Having liveness means that the communication delay of messages exchanged between a client and a replica, and between replicas, has an upper bound, but this upper bound may be unknown to the system. With this upper bound, messages exchanged within the replica network can be expected to eventually be delivered to the recipient, and the client can expect replies to its requests to arrive within a finite response time, without having to wait endlessly.

#### 2.1.1.3 Replica State

As mentioned in Section 2.1.1.2, the safety property requires that in a Byzantine fault-tolerant system the non-faulty replicas must keep their states consistent. More specifically, it requires that the *service state* of the replicated applications be consistent across all replicas, rather than the *system state* which is not synchronized, e.g., the information of the operating system and the middleware [17]. For a single replica, changes to its service state would affect the output of the service, while the system state may be different and not affect the output of the service by the replica. In the context of this thesis, we use the term *state* to refer to the service state of a replica for simplicity.

For the works discussed in this thesis, we assume a fully replicated system where the replicated services are identical across all replicas. As in most state machine replication systems, the replicated services are stateful and expose interfaces that allow clients to read and update the state of the service by issuing requests. However, we assume that application-specific knowledge might be required to decide which part of the state is affected by executing a particular request. Here we define the service state as a set of disjoint objects [23, 32] that can be monitored and accessed at runtime [23, 32, 56]. Mapping a piece of data to the service state should be possible after examining the service (e.g., from its source code). Based on this, we also assume that all objects can be divided into a set of non-overlapping partitions. Service instances must resemble a deterministic state machine, which requires that after executing the same sequence of inputs, but containing non-overlapping service partitions, all non-faulty replicas must produce the same sequence of outputs. State partitions can also be used to relax this so that non-interfering requests can be processed concurrently.

### 2.1.2 Fault Model

In this thesis, we present the proposed approaches assuming two fault models: the usual Byzantine fault model, as used in most traditional Byzantine fault-tolerant systems, and a hybrid fault model, where the system components have different resilience characteristics.

#### 2.1.2.1 Byzantine Fault Model

A common Byzantine fault model such as in [48, 23, 28, 57, 32] consists of at least $3f + 1$ replicas to tolerate up to $f$ Byzantine failures. Under this model, each component of the system is either non-faulty and operates correctly, or faulty and does not provide correct services. A faulty replica could fail in an arbitrary way, such as crashing or violating the protocol, but still continue to function. Furthermore, a faulty replica could even become malicious, so that it might try to prevent non-faulty replicas from following the protocol and delivering correct services. Since it is usually assumed that faulty replicas are uncorrelated and independent within a system, introducing diversity (e.g., by building a heterogeneous system environment and applying N-version programming [58, 45, 46, 12]) can reduce the fault dependency of different system components. In addition, Byzantine faults can also be caused by attacks: An attacker can launch an attack to gain full control over the compromised components of a system, causing them to behave maliciously.

In addition to faulty replicas, an arbitrary number of clients can also be faulty and even coordinated by an attacker to launch attacks. A common solution for dealing with faulty clients is to create a blacklist of clients that continuously exhibit faulty behavior, thus preventing their further access to the service. This blacklist can either be hosted locally or by a separate access control system. In this thesis we do not elaborate on how to deal with faulty clients.

#### 2.1.2.2 Hybrid Fault Model

In addition to the classical Byzantine fault model, where all components of a system are vulnerable to arbitrary failures, we also assume a hybrid fault model [35, 59, 60, 61, 34, 62, 63], where a system is a collection of components with different resilience characteristics.

The correctness of the functionality of these components is guaranteed by using a *trusted execution environment* (TEE) to perform the operations. A TEE is a secure, integrity-protected processing environment consisting of memory and storage capabilities [64]. Various definitions of TEE can be found in the literature to explain this notion: (1) In Terra [65], the TEE is described as a "dedicated, closed virtual machine isolated from the rest of the platform. Through hardware memory protection and cryptographic protection of storage, its contents are protected from observation and tampering by unauthorized parties." (2) According to GlobalPlatform [66], the TEE is "an execution environment that runs alongside but isolated from the device main operating system. It protects its assets against general software attacks. It can be implemented using multiple technologies, and its level of security varies accordingly." (3) In Trustworthy Execution on Mobile Devices [67], the TEE has "a set of

features intended to enable trusted execution: isolated execution, secure storage, remote attestation, secure provisioning and trusted path."

In this thesis, we assume that the system components running with TEE can either work correctly or fail by crashing, so that they produce trusted results that can be trusted by the rest of the system. Note that we need to ensure trustworthiness by implementing the TEE with trusted hardware and software, while minimizing the size and complexity of the trusted computing base (TCB) to protect it. Apart from the TEE, all other components of the replicas and the network in the system can fail in arbitrary ways, hence they do not trust each other. By using the TEE, the total number of servers required in a Byzantine fault-tolerant system under the hybrid fault model [60, 61, 34, 62, 63, 59, 35] can be reduced to $2f + 1$ replicas.

### 2.1.3 Background on BFT Systems

Despite different implementations, most existing Byzantine fault-tolerant systems [22, 23, 25, 28, 32, 59, 60, 68] are realized based on the same basic architecture, which contains two main parts: (1) the client side, where a set of clients issue requests to the replicas, and (2) the server side, where a set of replicas establishes an order of all requests through the *agreement stage* and enforces system consistency by executing requests according to the order through the *execution stage* [28]. Figure 2.1 illustrates the basic architecture at the conceptual level, while omitting the details of individual implementations.

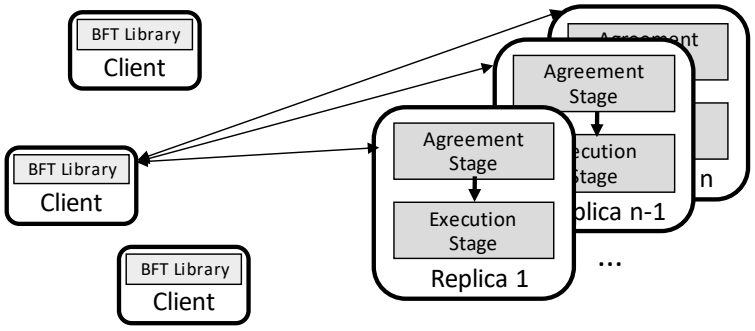



Figure 2.1: Basic architecture of a traditional BFT system.

#### 2.1.3.1 Client

First, we take a look at the client side. In this thesis, each client is referred to a physical or virtual machine where the client processes run. Each client interacts with server-side replicas and, by default, includes a *client-side BFT protocol library* that contains essential

functions such as issuing requests and validating replies by voting. This is due to the fact that a client of a BFT system cannot trust a single replica, as the replica could be faulty and ignore requests, or provide incorrect replies. Therefore, the client must verify the correctness of a reply from a replica by comparing it to other replies. The client can also establish and maintain connections to all replicas, which requires that it knows certain information about the server configurations, such as the identity of all replicas, the current view number, and the current leader replica.

A client can be recognized by a system-wide ID for its identity and embeds this ID in the request messages it sends to the replicas. In addition to the client ID, each client request is assigned a request number, which is derived from the client's local request counter. With these two aspects, each individual request can be identified with a unique *request ID*, so that we can ensure that duplicate requests are detected and discarded before they enter the agreement stage. In the execution stage, the request ID is also used to prevent a request from being processed more than once.

#### 2.1.3.2 Replica

In a typical Byzantine fault-tolerant system, the server side usually consists of a cluster of replicas, and within each machine runs a BFT protocol instance with an application. In this section, we explain the flow of processing a client request in individual stages: The agreement stage for the BFT protocol instance and the execution stage for the application, as well as the checkpoint mechanism.

**Agreement Stage** Once a leader replica accepts a client request, it first enters the agreement stage to be ordered by the BFT agreement protocol instance. The agreement instance then assigns a sequence number to the request. In this way, all requests are ordered by successive instances, eventually resulting in a totally ordered sequence of requests. As mentioned earlier (see Section 2.1.1.2), the safety property of a BFT agreement protocol is guaranteed by the fact that requests ordered into an identical sequence are executed across all non-faulty replicas. Even in the presence of faulty replicas, the safety property is not violated as long as the number of faulty replicas does not exceed the upper bound of tolerable faults.

In Section 2.1.4 we will take the basic BFT protocol PBFT [68] as an example, to explain the details of how to trigger an agreement protocol instance to order a request. By successfully executing the agreement instance, we can establish a stable and reliable association between requests and their sequence numbers. At the end of the agreement stage, the ordered requests enter the execution stage.

**Execution Stage** The execution of an ordered request can be done individually in the application of each replica. As with the agreement stage, the safety of the execution can be guaranteed as long as the number of faulty replicas does not exceed the upper bound of tolerable faults. In addition, the multiple executions must ensure consistency across all replicas so that their application states remain consistent and do not diverge. To achieve this, the

replicas must implement a deterministic state machine to ensure that the non-faulty replicas produce the same sequence of replies as outputs given the same sequence of ordered requests as inputs. Therefore, after each execution of an ordered request, the application state should be identical across all non-faulty replicas.

Recent research [23, 56, 69, 70, 71] has explored the potential of executing ordered requests in a deterministic order or in parallel rather than sequentially, while still maintaining consistent state across all non-faulty replicas. It has been proven that this approach can significantly reduce the performance overhead introduced by sequential executions.

**Checkpoint** In Byzantine fault-tolerant systems, all prepared and committed messages are logged in memory during runtime. During the execution stage, all non-faulty replicas periodically exchange messages to truncate the current application state and store it as a stable log on each local replica, designating it as *checkpoint*. In various implementations, this is usually done by taking a snapshot of the current state after a deterministically defined period of time, e.g., after processing a certain number of client requests. A checkpoint is stabilized once it is confirmed as evidence by $2f + 1$ replicas, and can be used in the following cases when a new replica joins the replica cluster or a slow replica wants to catch up with the current state. It can also be used to perform garbage collection for the internal state of the agreement protocol in the agreement stage [57].

### 2.1.4 The PBFT Protocol

To gain a deeper understanding of traditional Byzantine fault-tolerant systems, we will take a closer look at one of the most fundamental and classical BFT protocols, Castro and Liskov's Practical Byzantine Fault Tolerance (PBFT) [68]. We will use it as an example to briefly introduce the architecture of traditional BFT protocols with a protocol analysis in detail.

As a BFT protocol that assumes a Byzantine fault model and requires no additional components (e.g., a trusted execution environment), PBFT relies on a total of $3f + 1$ replicas to tolerate up to $f$ faulty ones. Among all replicas, a *leader* replica is chosen to guide the execution of a BFT protocol instance, and is responsible for proposing an agreement on the order of each accepted request. The remaining replicas, namely the *followers*, accept the proposed request order by participating in the agreement process. In addition to establishing the order, all replicas maintain a counter for the *view* and share it among themselves. Within each view, if followers suspect that the current leader is exposing faulty behavior, a leader replacement procedure called *view change* is issued and executed. When a quorum of all replicas have agreed to the view change, a new leader is chosen for the next view.

In a Byzantine fault-tolerant system in which the PBFT protocol is implemented, communication between clients and replicas, and the overall process of handling a request, involve the following steps (see Figure 2.2): ① In the REQUEST phase, a client sends a request to the leader. ② The protocol enters the next phase called PREPREPARE, and once the leader receives the request, it proposes the request with a sequence number to its followers. ③ In the following phase PREPARE, all followers multicast the proposal, and they can detect if a leader has sent different proposals to different followers. If a replica has received $2f$ match-

ing PREPARE messages (including its own) for a PREPREPARE message, then it considers the PREPREPARE message to be *prepared*. This way, followers can confirm to each other that at least $2f + 1$ replicas have seen the same request proposal sent by the leader. If this is not the case, a view change can be issued by the followers to suspect the current leader for its faulty behavior. This also ensures that at least $f + 1$ replicas are not faulty and they will not consider a different request for the current sequence number on their local view. ④ Finally, to confirm and learn each other's local views, in the next phase COMMIT, each replica broadcasts a COMMIT message and once $2f + 1$ COMMIT messages (including its own) have been received for the same request, the replica confirms that this request is *committed*. This guarantees that the local view is shared with each replica and, more importantly, that a majority of non-faulty replicas have prepared the same request. At this point, the safety property is guaranteed when a replica executes the request and returns the corresponding reply to the client. ⑤ In the final REPLY phase, once the client receives $f + 1$ matching reply messages from different replicas, it validates the correct result, since at least one reply message must be provided by a non-faulty replica.

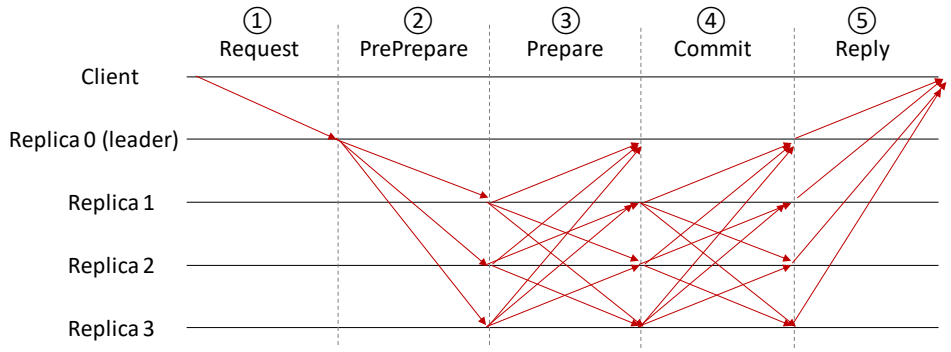



Figure 2.2: Message exchanges in PBFT.

Optimization is possible if the read and write requests are easily distinguishable. For a read request, $2f + 1$ replicas are queried to execute the request directly without ordering and return the replies. Only if the queried replicas can provide matching results does the client use the result, otherwise it restarts the process by resending the read request, and forcing a regular procedure. However, due to different processing speeds or the presence of faulty replicas, the returned results may differ, which often prevents successful read optimization.

## 2.2 Background on Cloud Computing

Cloud computing [2, 1] generally refers to a model that provides a shared pool of configurable, on-demand compute resources (e.g., applications, services, networks, storage). When the resources are needed by clients, they can be provisioned and delivered quickly, without much management overhead. Once the required resources are no longer needed, they can be immediately released and returned to the shared pool and reassigned to other

customers. Usage of the shared resource pool is automatically controlled and monitored by the cloud provider; customers can therefore refer to the reports provided for their payments.

In this section, we give a brief introduction to the most common types of cloud computing models. Depending on the model of the service provided, cloud computing can generally be classified into the following types, as shown in Figure 2.3:

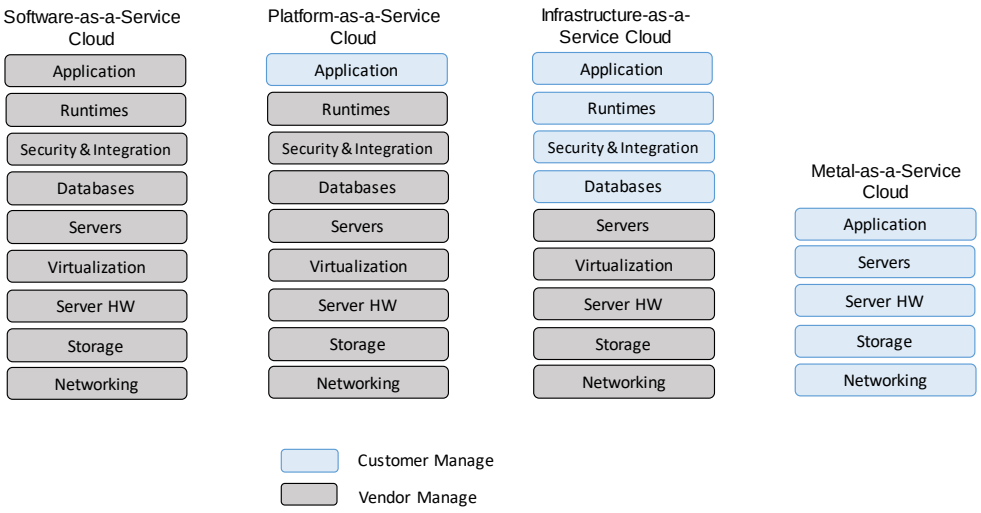



Figure 2.3: Structure of different types of cloud computing models.

### 2.2.1 Software-as-a-Service Cloud

The Software-as-a-Service (SaaS) model allows customers to use the provider's application software hosted on a cloud infrastructure, such as Gmail [72] and Dropbox [73]. SaaS cloud vendors manage the infrastructure and ensure that applications are easily accessible from various client devices. Customers of a SaaS cloud service are not bothered with administrative overhead except for limited configuration settings of custom applications.

### 2.2.2 Platform-as-a-Service Cloud

Platform-as-a-Service (PaaS) cloud vendors offer software developers a complete development environment in the form of a computing platform, including operating systems, programming language environment, database, and web server. Customers, such as software developers, can use the toolkit and other resources provided by a PaaS cloud to develop and run their applications on the platform, without incurring costs to purchase and manage the underlying hardware and software layers. Well-known PaaS cloud products such as Windows Azure [74], Google App Engine [75], Red Hat OpenShift [76] can guarantee that

the compute and storage resources required by the deployed applications are automatically adjusted as needed to avoid manual management by customers.

### 2.2.3 Infrastructure-as-a-Service Cloud

An Infrastructure-as-a-Service (IaaS) cloud allows customers to deploy and run any software [2], including operating systems and applications, on the virtual machines provided. Customers can deploy their software on a virtual machine and have control over resources such as operating systems, storage, and (partial) networking without having to manage the underlying hardware infrastructure. Instead, IaaS cloud vendors use a hypervisor, such as Xen [77], KVM [78], VMware Workstation [79, 80], to manage the infrastructure and run the virtual machines as guests, as with Amazon Web Services [81] and Red Hat OpenStack [82].

### 2.2.4 Metal-as-a-Service Cloud

Similar to IaaS cloud products, a Metal-as-a-Service (MaaS) cloud is able to dynamically provision computing environments and resources in machines. However, instead of virtual machines and shared compute resources, a MaaS cloud offers the flexibility to provide physical servers to each individual customer, without impacting the compute resources of other customers or custom applications. It also aims to automate the provisioning and deployment of physical machines to reduce the management burden on cloud providers, especially for large-scale cloud services.

### 2.2.5 Comparison of Different Clouds

We compare these cloud computing models to find out suitable options that can be used to implement the cloud environment proposed in this thesis, based on the following criteria:

- Low management overhead. In the IaaS model, customers are responsible for controlling the acquired compute resources, including the installation of operating systems and the runtime environment, which implies a high overhead for managing the resources and the deployed applications, resulting in a non-negligible overhead. The PaaS model can provide a platform, including system and application software, with minimal infrastructure operation and management requirements for the customer. Similarly, the MaaS model can also provide resources to customers in the form of physical machines with system and application software installed.

- Low complexity of custom application deployment. In the SaaS model, customers can just use the provided application instead of deploying their own applications. And in the IaaS model, while it is possible to deploy any software, the essential runtime environment must be managed by the customer. As with the PaaS and MaaS models, the deployment of customer applications is well supported by toolkits such as orchestration and automation tools, to significantly reduce the complexity.

As a result, we consider the PaaS and MaaS models as potentially suitable options for building a prototype of the proposed cloud environment in Chapter 6. We will specifically focus on these two models and disclose more details in Chapter 6.

# 3

# Problem Statement and Proposed Approach

In this chapter, we first analyze traditional replication protocols to identify the problems in integrating Byzantine fault-tolerant protocols with Internet services deployed in a cloud environment. Then, we summarize the issues as a problem statement and propose our approaches to solve them. Finally, we give a breakdown on each aspect of the proposed approaches and show the goals we want to achieve by applying the solutions.

## 3.1 Problem Analysis

In this section, we will analyze the consequences of integrating the state-of-the-art Byzantine fault-tolerant systems into a cloud environment for higher reliability of the services deployed in the cloud. First, we analyze the problems by taking a closer look at the architecture of traditional Byzantine fault-tolerant (BFT) replication protocols.

### 3.1.1 Traditional BFT Protocols

As described in Section 2.1.3, clients in most traditional Byzantine fault-tolerant systems [68, 23, 25, 26, 27, 35] must normally maintain connections to all server replicas. By sending requests to and receiving replies from all replicas, a client must not only implement the client-side protocol of the service, but also integrate a voting component to guarantee the safety of the service [68, 23, 25, 26, 27, 35]. This is because a client in a BFT system cannot simply trust a single replica, as it can be faulty and engage in arbitrary behavior, such as ignoring requests or generating faulty replies, and it must rely on the voting component to outvote incorrect replies. For this reason, each client establishes connections with all replicas in the system, rather than communicating with only one of them at a time.

Although the voting component helps clients verify the correctness of a result by comparing reply messages of different replicas, it still requires that a client obtains some knowledge about the replicas, such as their identities, in order to distinguish them, and this knowledge is typically provided to the client at configuration time. Many BFT systems use this knowledge to establish a dedicated shared secret between each client and each replica. The knowledge is then used to authenticate the exchanged messages, allowing a client to verify that a received reply is indeed from the presumed replica.

### 3.1.2 Single-leader Bottleneck

Despite the fact that all server replicas can interact with clients by receiving requests and sending replies, only one of them is selected to act as the leader at a given time within a view, e.g., either for the total order of requests [23] or for the execution preparation [31]. This is necessary to maintain a total order for all requests, so that the execution order of multiple replicas can be enforced.

To overcome the limitations of sequential execution and exploit the potential of multi-threading with multi-core hardware, systems have been proposed that allow non-conflicting requests to be processed in parallel [23, 26, 31]. In this thesis, we refer to such requests as independent, which applies, for example, to requests that operate on different parts of the service state. On the other hand, dependent requests must still be executed sequentially, otherwise they may even lead to inconsistent states on non-faulty replicas.

Parallel execution can offer great benefits for application scenarios where the actual request processing time is not negligible. However, it does not reduce the risk of a system performance bottleneck. Indeed, since existing BFT systems use only a single replica as a leader at a time, this slows down the processing speed of the entire system. To some extent,

this problem can be mitigated by rotating the leader role among replicas [25, 83]. However, this solution still relies on a totally-ordered sequence of requests and only rotates the responsibility for establishing the sequence, the effect is limited.

### 3.1.3 Non-transparent Access

Byzantine fault-tolerant state machine replication protocols mainly provide solutions to meet the requirements of replicated services for high availability and resilience against arbitrary faults. While BFT protocols were initially considered impractical, the seminal work of Castro and Liskov [48] enabled a stream of research that improved performance, lowered complexity, and reduced resource consumption of BFT protocols [27, 26, 23, 24, 25]. As it stands today, BFT protocols can be considered ready for custom deployments and are, for example, currently being evaluated in the context of permissioned blockchain network infrastructures [84].

However, most systems and services in production today are either unable to tolerate faults or are only resilient to crashes, leading to failures or undesirable behavior in situations where Byzantine failures actually occur [5, 85, 86]. The reason for the low adoption of BFT protocols in production is that, for user-facing offerings in open and heterogeneous environments such as the Internet, BFT protocols face a major hurdle that has been largely overlooked: client-side implementation. Standardized protocols such as HTTP and IMAP dominate most popular Internet services, and users are typically offered different implementations (e.g., different web browsers and email applications). This makes the integration of such protocols with BFT systems almost infeasible, e.g., to build a BFT-enabled web server, since BFT systems rely on the assumption that a client embeds a client-side BFT library and uses it to communicate with multiple replicas as well as to perform a majority vote on the received replies to overrule the incorrect replies. One possible solution is to extend the HTTP protocol and add custom software to browsers [87], but this would only address one of many standardized protocols.

Moreover, as shown in Figure 2.1, in BFT systems a client must maintain multiple connections to the replicas, e.g., one connection for each replica. On the one hand, this can be very costly for the clients, since migrating a cloud service from a non-replicated system to a BFT system involves increased network and processor utilization at runtime, since the client needs to receive, authenticate, and compare multiple replies for each operation. Such an increase in resource usage is especially challenging for those clients associated with low-bandwidth connections or have limited processing power, e.g., client services running on mobile devices.

On the other hand, ensuring the reliability of multiple connections places a significant burden on the management of both clients and replicas. For example, in this case, clients need to handle reconnections, so they require servers to expose their internal configuration settings, which is not very easy if the overall system is not designed to expose such information publicly. In summary, integrating existing client implementations of Internet services with BFT systems leads not only to actual migration costs, but also to runtime overhead, which explains why this step has not been taken for many applications so far.

### 3.1.4 Manual Deployment and Evaluation

Although many advanced prototypes of BFT systems have been developed in the last two decades, deploying such a prototype and evaluating its performance is still very laborious and error-prone. Most prototypes have similar deployment requirements, such as running on a cluster of servers, but the composition of the cluster may differ significantly. For example, most of them can run Linux operating systems and have some requirements for specific runtime environments, while the impact of using different Linux distributions as well as different kernel versions is still unknown.

Therefore, to achieve optimized performance of a BFT system, continuous testing with different setups and configurations should be conducted in an efficient way, i.e., they should not be handled manually. It is well known that performing and validating a manual deployment process is often time-consuming, error-prone, and distracts engineers from focusing solely on development. Therefore, we point out the last problem that this thesis intends to solve: how to replace the manual deployment and evaluation processes of different BFT systems.

## 3.2 Suggested Approach

To address the above problems and challenges, we propose a novel design to (1) improve the performance of replicated services in a Byzantine fault-tolerant system that also (2) provides trusted and transparent access to the replicated services, and (3) a mechanism for automated deployment and evaluation in a cloud environment. In the following, we briefly present the proposed approach in three main parts. Note that these parts can be essentially combined, but this is not addressed in this thesis as it is orthogonal to the proposed solutions and therefore beyond the scope of the thesis.

### 3.2.1 Parallel Agreement and Execution

To solve the single-leader bottleneck problem, we propose SAREK, a parallel ordering framework that instantiates multiple single-leader-based BFT protocols independently. SAREK not only allows concurrent executions, but also exploits parallelism in the ordering of requests: It partitions the service state and linearly orders only those requests that access the same state partition(s), by creating a partition-specific schedule. In this way, SAREK can perform agreements for independent requests concurrently. Also, by selecting different replicas as the leader of different partitions, SAREK can distribute the workload induced by the leader role across all replicas. After the agreement is completed, a dedicated execution instance is responsible for processing the requests accessing each partition according to the local schedule.

### 3.2.2 Transparent Access

To enable transparent access to replicated services, we propose a system that achieves client-transparent Byzantine fault tolerance by moving the traditional client-side functions of a

BFT protocol, such as connection handling, request distribution, and majority voting, to the server side, which is in close proximity to the replicas. This is made possible by relying on a trusted subsystem that can only fail by crashing and implements basic message handling, majority voting, and transport encryption: the TROXY. At the implementation level, the TROXY leverages the trusted execution support provided by Intel's Software Guard Extensions (SGX) [88, 89]. At its core, SGX provides a set of new instructions that allow user-level code to allocate private and secure regions of memory called *enclaves*. By executing application code within enclaves, SGX provides CPU-enhanced application security and protects the enclaves from being manipulated by malicious privileged code or even hardware attacks such as memory probes. Therefore, the functionality of TROXY is guaranteed to be trustworthy even in the presence of Byzantine faults in the surrounding replicas.

### 3.2.3 Automated Deployment and Evaluation

By building an automated deployment and evaluation framework, we bridge the development environment with the production environment. This framework will be the fundamental part of a test environment for various BFT systems. To build the framework, we first need to prepare the platform on which the BFT systems will run. The normal and regular procedure involves purchasing a certain number of physical/virtual servers, installing operating systems with the required runtime environments, and finally deploying the BFT system. Instead of performing the entire process manually each time, we propose a mechanism based on a Metal-as-a-Service (MaaS) cloud that aims to automate all the steps of the procedure and provide flexible configuration options to meet the different needs of BFT systems. This mechanism consists of several parts: (1) an infrastructure platform that provides the hardware, e.g., a cluster of servers running BFT systems and replicated services; and (2) an automation tool that is integrated into the platform and is used for automated deployments and evaluations by launching predefined scripts. The platform and automation tool are managed by the administrator and infrastructure operators.

## 3.3 Objectives and Solutions

In the last section we briefly presented the proposed approach to address various aspects of the problems. In this section, we state the objectives we want to achieve in each aspect and explain how we can achieve them by further elaborating the proposed approach.

### 3.3.1 A Multi-leader BFT Protocol

In SAREK, after partitioning the service state, it is a simple operation for a request to access only a single partition. However, it requires additional overhead to support requests that operate on multiple partitions ("*cross-border requests*"). For example, despite being ordered in multiple partition-specific schedules, a cross-border request must *not* be executed more than once. In addition, the processing of a cross-border request must be consistent across

all schedules determined by all partitions affected by the request. SAREK satisfies these requirements by using a mechanism based on a combination of partition prioritization and safe reordering of requests. It ensures consistency by only allowing the execution instance of the partition with the highest priority to actually process the cross-border request, while putting the instances of the other partitions involved on hold in the meantime.

SAREK relies on application-specific knowledge to define service-state partitions and predict which partitions will be affected by a request. To this end, each replica has a deterministic PREDICT() function that identifies the state objects that need to be read or written when processing a particular request. Since implementing a precise PREDICT() function may not be feasible (or considered too costly) for some applications, e.g., because the set of objects accessed depends on the internal service state, SAREK provides support for handling imprecise knowledge up to the point where mispredictions are handled. During execution, the system monitors accesses to each state object and consequently detects when a request attempts to operate on a partition that is not included in the output of the PREDICT() function. When such a misprediction occurs, SAREK initiates a new prediction on the request and safely updates the schedules of the affected partitions, preventing any form of rollback.

In addition to SAREK, DYPART abstracts each application's access pattern to the state objects to create an application-specific request dependency. Based on this, it applies a deterministic partitioning algorithm to further improve the quality of state partitioning and reduce synchronization overhead across partitions.

### 3.3.2 Trusted Proxy in BFT Systems

TROXY provides a trusted proxy to ensure that clients can access replicated services over the original legacy protocols without modification, using trusted execution technology to ensure security and reliability. Once a TROXY instance receives a client request, it forwards the request to the underlying BFT protocol, which in turn orders the request, executes it, and forwards the computed replies to the requesting TROXY. Once the responsible TROXY instance has received enough replies, it performs a vote on the replies and returns the correct result to the client.

Since a malicious replica can intercept the communication of its TROXY, we ensure that the replica cannot modify messages without being detected. Communications between clients and TROXY instances are protected over secure, encrypted connections, which are the norm for an increasing number of Internet-based services [90]. In addition, messages exchanged between TROXIES and replicas are authenticated using common message certificates, as is the norm for BFT systems. While immune to arbitrary or malicious behavior, it is still possible for a TROXY instance to crash or become disconnected from its clients and become unavailable as a result. This case is equivalent to a failing service replica in commodity infrastructures and can be handled by DNS round-robin or load-balancing applications that allow failover to another TROXY instance.

### 3.3.3 BFT Systems with Metal-as-a-Service Cloud

Both traditional operation system virtualization and emerging container systems aim to increase the utilization of a single physical machine, e.g., via multi-tenant Platform-as-a-Service (PaaS) clouds. Metal-as-a-Service (MaaS) takes a different approach. Instead of virtual machines or containers, MaaS clouds offer customers complete physical machines. They are particularly useful when the software deployed has special features that cannot be virtualized (or only at great expense) because it is deployed directly on the hardware. Also, since customers have full control over the host machines, they can always use the entire compute resource for compute-intensive workloads without interfering with other customers applications. Therefore, we consider the MaaS cloud model as a good option for building the infrastructure of the proposed test environment. Based on this, management and configuration tools are also needed to automate the deployment, evaluation and result analysis of various BFT systems. This can be done by creating and executing predefined scripts with various configuration options. Running the configured scripts provides a lot of convenience when it comes to managing and orchestrating of the BFT protocol and service replicas, as the entire process can be fully automated.

# 4

# A Multi-leader-based BFT System

Recently proposed Byzantine fault-tolerant (BFT) systems have achieved high throughput by processing requests in parallel [23, 26, 31]. However, since their agreement protocols still rely on a single replica (i.e., the leader) to initialize the ordering process, it is a large overhead to establish a global total order for all requests before executing them, which limits their performance and scalability.

In this chapter, we present the first aspect of the solution provided in this thesis: SAREK, a parallel ordering framework that leverages service state partitioning to exploit parallelism during both the agreement and execution stages. Service state partitioning is based on request dependency abstracted from application-specific knowledge. Therefore, instead of a leader at a time for the entire system, SAREK uses a leader *per partition* to specify an ordering for requests accessing the same partition. SAREK supports operations that span multiple partitions and provides a deterministic mechanism to process them atomically. In case there is not enough application-specific knowledge to determine which partition(s) a request will access, SAREK also provides mechanisms to handle mispredictions without requiring rollbacks.

To improve the quality of state partitioning and reduce performance overhead, we also introduce Dynamic State Partitioning (DYPART) in this chapter. DYPART is an additional framework to SAREK that periodically reconfigures state partitioning for different usage patterns, relying on knowledge about the relationships between state objects obtained by collecting request dependencies. It uses a powerful graph partitioning algorithm to ensure that the resulting state partitions can achieve both balanced workload and low synchronization between partitions.

## 4.1 A Parallelized BFT System

In distributed systems, it is a general and common method to use state machine replication to implement fault-tolerant service by replicating servers and coordinating client interactions with server replicas [91]. In state machine replication, requests issued by clients are first put into a total order using an agreement protocol and then executed sequentially on multiple replicas. Depending on the protocols used in the system, the replicas are able to tolerate crash-stop [92, 93] failures or Byzantine failures [68]. In the context of Byzantine fault-tolerant (BFT) protocols, several research efforts have been proposed during the last decade to increase the practicality of using such protocols in distributed systems by improving their performance. For example, some protocols allow parallel processing of non-conflicting requests [23, 26, 31] to overcome the limitations of sequential execution and to exploit the potential of multi-core hardware. In this thesis, such non-conflicting requests are called *independent* requests, meaning that they operate on different parts of the service state during their execution time. Therefore, they can be executed in parallel since they do not cause conflicts on the state of the same data. In contrast, *dependent* requests must still be executed sequentially, otherwise the service states of the non-faulty replicas could diverge and become inconsistent.

Although parallel execution in BFT systems offers great benefits for application scenarios where the actual request processing time is not negligible, it does not reduce the risk of system bottleneck that is caused by using a single replica as a leader. In fact, most existing BFT systems use a single replica for either request ordering [23] or execution preparation [31], which actually slows down the processing speed of the system. To some extent, this problem can be mitigated by rotating the leader role among all replicas [25, 83]. However, since these approaches are still based on a totally-ordered sequence of requests and only distribute the responsibility of establishing such a total order to each replica, their impact is limited.

In this chapter, we introduce SAREK, a parallel ordering framework that runs instances of multiple single-leader-based BFT protocols independently. SAREK allows concurrent request executions and also exploits parallelism in request ordering. With SAREK, the service state of a BFT system is partitioned, and only requests accessing *the same* partition(s) are ordered linearly according to a partition-specific *schedule*. In this way, the ordering of independent requests can be performed concurrently without diverging service state. Moreover, by having multiple leaders, each responsible for one of the partitions, the system is also able to distribute and balance the workload caused by the leader role across all replicas. Once the agreement process is completed, each partition has a dedicated execution instance responsible for processing the ordered requests according to that partition's execution schedule.

While handling a request that accesses only a single partition is straightforward in such an environment, additional efforts must be made to support requests that operate across multiple partitions, called "cross-border requests". Since a cross-border request accesses data from multiple partitions, it should be ordered by all appropriate leaders in their individual schedules. However, despite the multiple ordering, a cross-border request must not be executed more than once and, in addition, its processing must be consistent across the schedules of all affected partitions to ensure consistency of service state. To meet these re-

quirements, SAREK uses a mechanism based on partition prioritization, so that the execution of a cross-border request is performed only by the execution instance of the partition with the highest priority among all involved partitions, while the instances of other partitions are put on hold in the meantime.

SAREK relies on application-specific knowledge to define the service state partitioned and *predict* the partitions to be accessed by each request. To achieve this, each replica leverages a deterministic PREDICT() function to identify the state objects that will be read or written during the processing of a particular request. However, implementing a precise PREDICT() function may not always be feasible for some applications or may be considered too costly, e.g., because the set of objects accessed is unknown or depends on internal service state. In such cases, SAREK also offers solutions to deal with mispredictions caused by imprecise knowledge. It monitors any access to state objects during the execution of a request, and consequently detects when a request tries to operate on a partition not defined by the PREDICT() function. When such an attempt is detected, SAREK halts executions, initiates a re-prediction of the request and then safely updates the schedules of the corresponding partitions, preventing any form of rollback.

In this thesis we also show that it is possible to implement SAREK with an existing BFT implementation without requiring modifications to the most complex part: the agreement protocol. Instead, the agreement stage process can be treated as a black box and instantiated multiple times, one for each partition. This makes most existing BFT agreement protocols compatible with SAREK. We evaluated our prototype with a microbenchmark and the YCSB (Yahoo! Cloud Serving Benchmark) [94] benchmark, which not only provides operations for accessing single objects, but also allows clients to issue requests that access multiple objects atomically, leveraging SAREK's support for cross-border requests.

## 4.2 Related Works

### 4.2.1 State Machine Replication

In traditional BFT state machine architecture [68, 57, 32], the agreement stage is usually combined with the execution stage as shown in Figure 4.1a. In such systems, the function of agreeing on a linearizable order of all requests is tightly coupled with the function of executing these requests on all state machine replicas. In [28] Yin et al. present a new architecture that separates these two functions as shown in Figure 4.1b. This architecture is based on the observation that the agreement stage of the traditional architecture produces a cryptographically-verifiable proof of the order of a request that can be verified by any server. Therefore, it is possible for the execution stage to be separate from the agreement stage, e.g., by running different processes or nodes.

SAREK features separation of the agreement and execution stage and provides parallelism in both stages. Related works aimed at improving the scalability and throughput of each stage are discussed below.

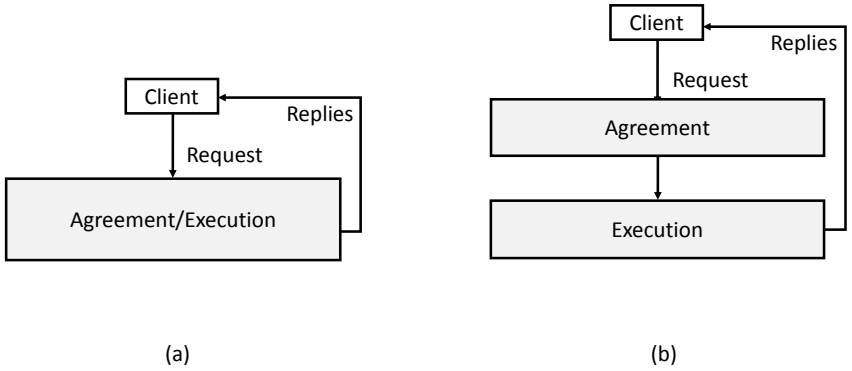



Figure 4.1: High level architecture of (a) traditional BFT state machine replication and (b) separate BFT agreement and execution.

#### 4.2.1.1 Agreement Stage

For a state-machine system that tolerates crash-stop failures, the throughput of agreement can usually be improved by generalizing the consensus measures. For example, Generalized Paxos [40] relaxes consensus from agreeing on a single request to agreeing on a partially ordered set of requests. It is applied to handle concurrently issued but non-interfering requests, where the execution order does not matter, to ensure that they can always be executed in two message delays. EPaxos [41] is a crash-tolerant protocol that allows replicas to agree on requests without requiring a designated leader. EPaxos only orders those requests that interfere with each other. Compared to SAREK, Generalized Paxos and EPaxos address no tolerance to Byzantine faults. EPaxos offers the best performance with low contention because the replica must delay executing a command until it receives commit confirmations for the command's dependencies.

As with BFT systems, view (leader) change is typically used in the agreement stage to increase the robustness and load balancing of the system and ultimately improve throughput. Introducing parallelism by running multiple agreement instances simultaneously is also commonly used in both crash-stop and BFT systems to improve scalability. Aardvark [37] minimizes the negative impact that a malicious leader can have on system performance by allowing the other replicas to monitor its performance. If the leader fails to propose new requests within a certain period of time, which is gradually reduced, another replica takes over the role and becomes the new leader. In Spinning [25], the change of leader occurs even more frequently than in Aardvark, namely automatically after each request is processed. This approach has the advantage of distributing the extra load associated with the leader role across all replicas. Nevertheless, the single-leader bottleneck remains, since the system must maintain a total order of all requests.

RBFT [95] relies on multiple concurrent BFT agreement instances for robustness so that all requests from all agreement instances are totally ordered, but are only executed by a dedicated master instance. That is, RBFT introduces parallelism in the agreement stage by fully replicating the entire agreement process, and thus also suffers from the single-leader

bottleneck. SAREK, on the other hand, partitions the agreement stage and uses multiple leaders to improve performance. Farsite [96] is a large-scale distributed file system that is resilient to Byzantine failures. The system achieves scalability by partitioning the service state and rarely coordinates when more than one partition is affected (e.g., for rename operations). Unlike SAREK, Farsite relies on multiple dedicated BFT clusters, each running an independent BFT agreement protocol. Compared to SAREK, Farsite does not address parallelization of request processing in the agreement and execution stages for a large number of applications, nor does it provide support for dealing with imprecise knowledge about the application. COP [27] achieves high scalability in BFT systems by executing successive consensus instances in parallel with independent pipelines. However, it is still based on a single-leader approach, but executes multiple rounds concurrently due to multi-threading.

P-SMR [42] achieves parallelism by using different multicast groups for requests that can be executed concurrently. The mapping of the requests to different multicast groups is based on service-specific semantics. In a follow-up work [43], Marandi et al. studied the impact of optimistically executing this mapping and proposed an approach that triggers rollbacks when inaccurate assumptions lead to inconsistencies across replicas. Unlike P-SMR, SAREK is not limited to crash-stop failures. Moreover, SAREK detects mispredictions and is therefore able to initiate re-predictions before replicas become inconsistent, thus avoiding rollbacks. S-SMR [44] achieves scalable throughput by partitioning service state and using caches to reduce synchronization across partitions. It relies on an atomic multicast to order requests and implements execution atomicity to guarantee linearizability. This approach must be adapted to tolerate Byzantine failures, eventually leading to a Farsite-like system.

#### 4.2.1.2 Execution Stage

Executing requests concurrently or in parallel is often used for execution stage throughput improvement. Kotla et al. [23] introduced a parallelizer module between the agreement stage and the execution stage that allows a BFT system to execute requests that do not interfere with each other in parallel. Similar to SAREK, such requests are identified based on application-specific knowledge. However, unlike SAREK, there is no parallelism in the agreement stage and a total order is established across all requests, not just the dependent ones. The same argument applies to ODRC [26], which achieves parallelism in the execution stage by processing each request on a subset of replicas rather than on all replicas. This allows the resources freed up on each replica to be used to execute more requests.

In EVE [31], the replicas do not agree on requests before executing them. Instead, the replicas first execute requests concurrently and then attempt to agree on the corresponding state changes. If this attempt fails, the replicas roll back and repeat the executions, but this time in a sequential order. EVE assumes that the majority of requests do not share data, or if they do, they do so in a limited way, so conflicts rarely occur. SAREK assumes that there are sets of requests that do share data and require sequential ordering accordingly, but these sets themselves usually do not interfere with each other, otherwise that can be detected and addressed, avoiding rollbacks entirely.

### 4.2.2 Distributed Transactional Systems

Granola [97] presents a coordination infrastructure for distributed transactions in a crash-stop model that provides strong consistency while reducing coordination overhead. This is guaranteed by using a timestamp-based coordination mechanism to achieve serializability of transactions that execute on a single storage node or across a set of nodes, but do not require agreement on execution order. With similar functionality to Granola, Calvin [98] provides high availability and complete ACID transactions in partitioned database systems. It achieves this by implementing a sequencing layer above the storage system to handle data replication, which executes a global agreement protocol on read and write transactions across all replicas. According to the generated deterministic locking order, it deploys a transaction scheduling layer that serve as concurrency control. Compared to these two systems, SAREK is initially developed for a different fault model. Moreover, it does not enforce a total order for all requests, nor does it require transactional semantics to handle conflicts.

## 4.3 The Sarek Approach

The main goal of SAREK is to distribute the extra workload associated with the leader role in traditional single-leader based agreement protocols [68, 57, 32, 37] across all replicas, thereby enabling parallelism in both the agreement and execution stages of request processing. It is based on the observation that most existing single-leader BFT systems conservatively assume a total order for all requests, but this is far too pessimistic for many applications. For example, in applications such as key-value stores or web applications, not all requests interfere, but only a fraction of them. Therefore, only the dependent requests that access a common state should be ordered with respect to each other.

Consequently, SAREK provides the idea of running multiple BFT agreement instances in parallel, each of them responsible for a fraction of the total service state and maintaining a partial order for the requests accessing the data from that fraction of the state. In fact, SAREK is inspired by *Generalized Consensus* [40] which shows that under the crash-stop failure model, consensus can be relaxed from agreeing on a single request to agreeing on a partially ordered set of requests.

### 4.3.1 Parallel Agreement and Execution

The establishment of parallelism in SAREK is based on the state partitioning. On this basis, each replica becomes a host of multiple BFT instances, and each instance is responsible for one partition of the service state. For example, in the error-free case, the leader role of each BFT instance is distributed as shown in Figure 4.2.

State partitioning is handled logically by determining to which partition a state object (i.e., an element of service state, see Section 2.1.1.3) belongs, so it is essential to acquire application-specific knowledge about service state. For example, in a key-value store application (see Section 4.4), where the state objects are key-value pairs, the service state of the system can be partitioned by simply dividing the key space. Based on this, a BFT agreement

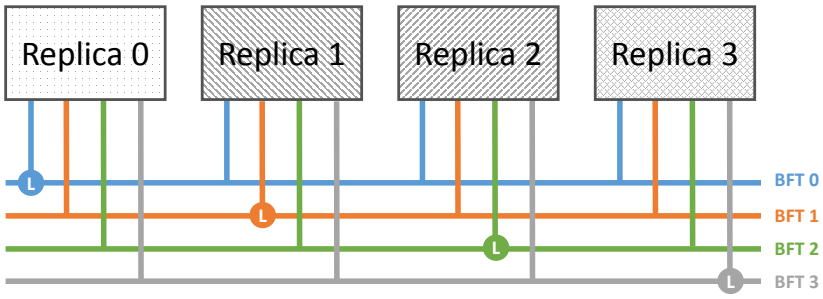



Figure 4.2: In the error-free case each replica is a leader for one of the BFT agreement protocol instances.

instance only takes the responsibility for ordering the requests that access its associated state partition(s) (e.g., keys belonging to the allocated part of the key space). Therefore, SAREK is able to order the requests per partition instead of "blindly" setting a total order for all requests.

Figure 4.3 illustrates the system architecture of SAREK. The agreement stage is the same as in a single-leader BFT protocol, to make SAREK compatible with common BFT protocols that feature a separation of agreement and execution stages. And to map client requests to the appropriate BFT instances, SAREK introduces a *predictor* component that hosts an application-specific PREDICT() function. Each non-faulty replica has its own local predictor, which is executed deterministically and is consistent with the predictors of the other replicas. The PREDICT() function analyzes each request based on its type, payload and runtime state to compute the state objects that will be accessed during execution, and finally specifies which partitions those objects belong to. Similar ideas regarding a request analysis component can also be found in some other research papers [23, 26, 32]. The implementation of such a predictor component relies on certain knowledge about the application, e.g., in a key-value store (see Section 4.4) keys are used for prediction because they contain the most specific information about the request.

The predictor component is executed before the agreement, which allows for coordinated request distribution (see Algorithm 4.1). At system startup, each replica $i$ is selected as the leader of a single BFT agreement instance. A client sends a request $m$ to all replicas, which then triggers the PREDICT() function at each replica to return a set of partitions that $m$ will access. How the prediction of multiple partitions works in detail is explained in the following section. Once the partitions are determined, each replica checks to see if it is the leader for one or more BFT instances responsible for the order of requests accessing the predicted partitions (line 8 - 10). If so, it then enters the agreement stage to order the request. As of now, we no longer distinguish between an instance and a partition, since they actually correspond one-to-one.

Once the agreement stage is complete, the request is passed to the execution stage as part of the partition-specific schedule (see Figure 4.3), which is the order of requests accessing

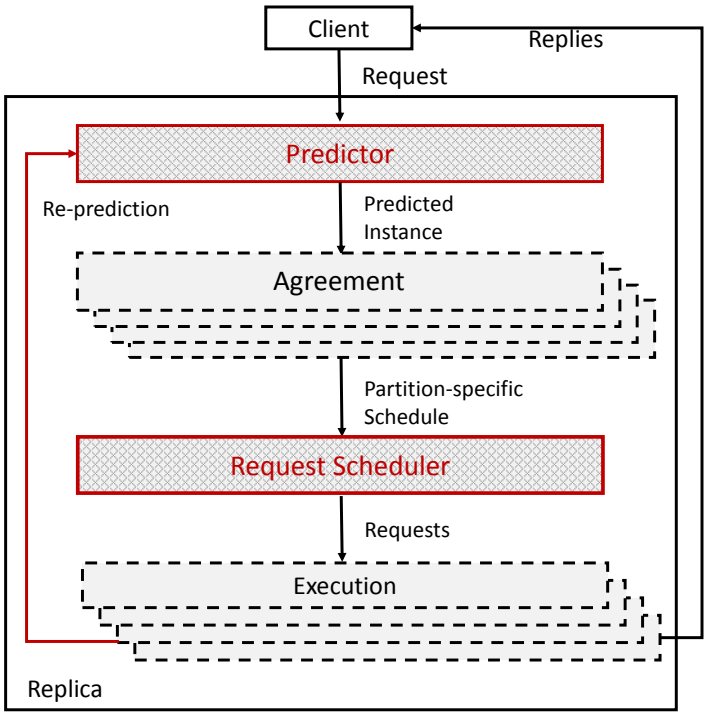



Figure 4.3: System architecture of SAREK.

the same partition. A new component placed between the agreement stage and the execution stage, namely the *request scheduler*, is responsible for ensuring that executions follow the correct order specified by the agreement stage. In the case of a *simple request* that only accesses a single partition, the agreement and execution processes are quite simple: The one and only replica that is the leader of the single partition initiates the ordering. Once agreement on the order is reached, all replicas pass the request to the execution stage. Meanwhile, access to the partition is monitored by the request schedulers to guarantee consistency across all non-faulty replicas.

### 4.3.2 Cross-Border Request Handling

When a request accesses multiple partitions, i.e., a *cross-border request*, in addition to the simple requests, further actions must be taken to ensure: (1) this request is ordered by all predicted BFT instances while (2) it is executed exactly once.

#### 4.3.2.1 Agreement Stage

To meet these two requirements, the treatment of a cross-border request takes an extra step: It "splits" (logically) into several *sub-requests*, where each sub-request actually contains the data of the original request and is supposed to be ordered by the corresponding instances.

**Algorithm 4.1** Predict and order requests

```
 1 initialize:
 2     bft_id as a BFT agreement instance
 3     P_mi as a set of predicted partitions

 5 upon receiving ⟨REQUEST, m⟩ at replica i do
 6     P_mi := PREDICT(m) // predict partitions
 7     for each par in P_mi
 8         bft_id := instanceOf(par)
 9         // get responsible instance
10         if i equals leaderOf(bft_id)
11             start agreement stage of ⟨REQUEST, m⟩
12         end if
13     end for
14 end
```

Consequently, the ordering of any pair of requests accessing the same partition is uniquely defined across all non-faulty replicas, based on the orders and corresponding partition schedules.

#### 4.3.2.2 Execution Stage

Upon completion of the agreement stage, all non-faulty replicas have identical schedules on their partitions and are about to enter the execution stage. In SAREK, the key to consistent executions is that of all the sub-requests of a cross-border request, only one of them should be executed, while the others act like placeholders that only mark the order of those sub-requests in their partition schedules. This requires deterministic selection of an instance as an *execution instance*, e.g., by prioritizing an instance to execute the sub-request. The non-execution instances put their sub-requests on hold and remove them only when the execution instance is complete. In our implementation, the execution instance is selected as the one with the smallest ID number among all the participating instances.

The request scheduler assigns a specific type to each request/sub-request before passing the request to the execution instance. If a simple request accesses only one partition, its type is marked as EXECUTE. Otherwise, a sub-request of a cross-border request is further distinguished as (1) the type CROSS-BORDER-EXEC to be executed by a deterministically selected execution instance, or (2) the type CROSS-BORDER-SYNC, which behaves like a placeholder. Requests are delivered and queued for execution after classification. In Figure 4.4, we show an example where the circled $R2$ represents the sub-request of type CROSS-BORDER-EXEC that is executed by the execution instance $T0$, while the squared $R2$ at $T1$ is of type CROSS-BORDER-SYNC that synchronizes only with the execution of $T0$. This procedure is summarized in Algorithm 4.2.

An execution attempt is triggered when a request reaches the head of the local partition schedule, which basically resembles a queue. Depending on the type of the request, execution is handled as follows: (1) A request of type EXECUTE can be executed immediately, since the execution is independent and does not interfere with other partition schedules (line 8).

**Algorithm 4.2** Instance at execution stage

```
 1 initialize:
 2     // array only used for resolving request cycle
 3     blocked_by := Array[no_of_partitions]

 5 while executing at instance bft_id do
 6     req := PartitionSchedules[bft_id].peek()
 7     if req.type is EXECUTE
 8         execute(req)
 9         PartitionSchedules[bft_id].dequeue()
10     else if req.type is CROSS-BORDER-EXEC
11         // check sync partitions notifies
12         if req.ready()
13             executeCrossBorderRequest(req)
14             PartitionSchedules[bft_id].dequeue()
15             // remove req's corresponding sync partitions
16             for all sync_partition in req's sync partitions
17                 PartitionSchedules[sync_partition].dequeue()
18                 blocked_by[sync_partition] := NULL
19                 notify instance sync_partition
20             end for
21         // detect request cycle
22         else
23             if detect_and_resolve_cycle(bft_id, req)
24                 goto 12
25             else
26                 blocked_by[bft_id] := req
27                 wait for corresponding CROSS-BORDER-SYNC
28                 continue
29             end if
30         end if
31     else
32         if req.type is CROSS-BORDER-SYNC
33                 exec_partition := the partition of corresponding CROSS-BORDER-EXEC
34                 blocked_by[exec_partition] := NULL
35                 notify instance exec_partition
36                 blocked_by[bft_id] := req
37                 wait for corresponding CROSS-BORDER-EXEC to finish req
38         end if
39     end if
40 end while
```

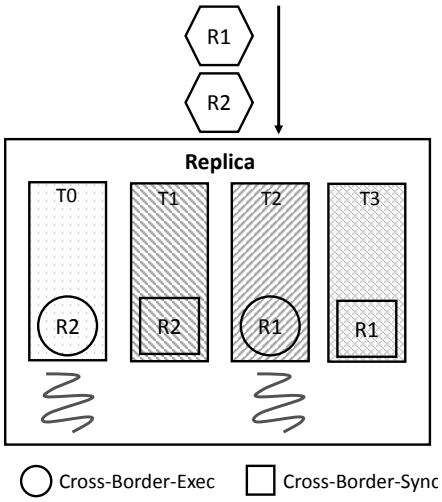



Figure 4.4: Distribute sub-requests of cross-border requests in SAREK.

(2) In the case of type CROSS-BORDER-EXEC, the request is first checked to see if all associated CROSS-BORDER-SYNC sub-requests are also at the head of their partition schedules (line 12). If so, the current instance executes the request and then removes all associated sub-requests from their partition schedules (line 13 - 20). Otherwise, it attempts to detect and handle request cycles, if any (line 23 - 30), which is discussed in detail later. (3) For a request of type CROSS-BORDER-SYNC, the instance notifies the corresponding sub-request of type CROSS-BORDER-EXEC and waits until the execution of the sub-request CROSS-BORDER-EXEC is complete (line 32 - 39). This ensures system consistency by forcing the execution instance to execute a single request only once.

#### 4.3.2.3 Request Cycle

As mentioned earlier, the partition schedules of different instances may differ, e.g., due to the arrival of some out-of-order requests, since there is no deterministic delivery. Figure 4.5 shows an example of such a case, where requests $R1$ and $R2$ arrive at $Replica0$ and $Replica1$ in different order. Assume that these two replicas hold the leader roles for instances $T0$ and $T1$ respectively, then the sub-requests of $R1$ and $R2$ will be ordered differently in the partition schedules of the two instances. According to Algorithm 4.2, the CROSS-BORDER-EXEC sub-request of $R2$ should wait for the notification of the instance that processes the CROSS-BORDER-SYNC but blocks the CROSS-BORDER-EXEC sub-request of $R1$. Meanwhile, the CROSS-BORDER-SYNC of $R1$ is put on hold waiting for its CROSS-BORDER-EXEC to execute while the CROSS-BORDER-SYNC of $R2$ is blocked. As a result, a *request cycle* occurs as both instances are blocked and waiting for the other. It is more likely to detect such a situation when more than two partitions are involved.

Such request cycles can be detected and resolved using the request scheduler as shown

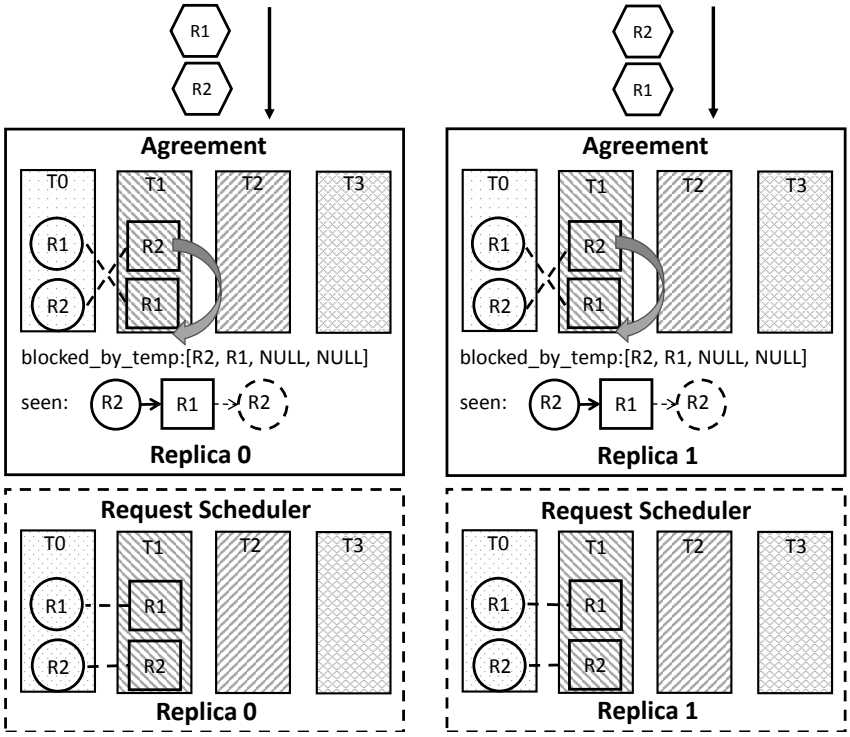



Figure 4.5: Cause of a request cycle and the solution to it.

in Algorithm 4.3. It is called when an instance holds a CROSS-BORDER-EXEC and waits for notifications from its corresponding CROSS-BORDER-SYNCs, and the procedure is protected by a mutex lock. To resolve a request cycle, the first step is to find out which instances are really blocked as part of a request cycle, that is, have no chance of being resumed. This detection relies on an array $blocked\_by$ that contains the currently waiting instances. At the beginning of Algorithm 4.3, a snapshot of $blocked\_by$ is taken. However, since this array changes during runtime, we need to recursively collect the free instances that are not blocked in the snapshot or still have a chance to finish (line 17 - 24). In this way, we may overestimate the free instances but, more importantly, we will not miss any blocked instances, according to the following lemma. We prove the correctness of all lemmas in Section 4.5.

**Lemma 1.** *For any partition $bft\_id \in \mathcal{B}$ determined by line 25 in Algorithm 4.3, the $blocked\_by[bft\_id]$ will remain the same status as in the snapshot and not change unless the request cycle is resolved.*

After all blocked instances are known, the algorithm starts detecting request cycles. It uses the linked list *seen* to store the requests that block their instances. For a given item in *seen*, it must wait until the next item finishes, before it can execute. To ensure that the linked list encounters a loop at the end, we have the following lemma:

**Lemma 2.** *The while-loop starting at line 33 in Algorithm 4.3 will terminate.*

We also show that the next element is found deterministically:

**Lemma 3.** *In the cycle detection of Algorithm 4.3, line 37 will always deterministically find the next node for any node in seen. The next node is determined only by the ordering result of each partition, but is irrelevant to race conditions at runtime.*

The final step is to resolve the detected cycle. We deterministically select the request at the head of the instance that has the smallest ID (line 45), and let that instance execute first. Since both cycle detection and resolution are handled in a deterministic way and are irrelevant to race conditions, we can derive the following theorem:

**Theorem 1.** *Consistency between any non-faulty replicas can be guaranteed in Algorithm 4.3.*

The request cycle shown in Figure 4.5 can then be solved by adjusting the order of CROSS-BORDER-SYNC $R2$ in their partition schedule, in this case by moving it to the head of the schedule. In this way, the blocked CROSS-BORDER-EXEC $R2$ can immediately stop waiting and resume its execution. Changing the order of scheduled CROSS-BORDER-SYNC is possible due to the fact that only the CROSS-BORDER-EXEC of a cross-border request is uniquely and deterministically defined in its schedule. As long as all replicas follow this rule and perform the same operation, the service state remains consistent. As far as the detection and resolution of request cycles are completed, all requests of the adjusted partition schedules can be executed.

### 4.3.3 Fault Handling

Now we move on to the fault handling of SAREK, which generally relies on two things: (1) the general fault handling mechanism provided by the BFT protocol itself, and (2) additional measures to handle faulty behavior revealed by the predictor. Here, we first present the fault handling mechanism introduced by SAREK, including fault detection and recovery methods, and then provide insight into how to keep the complexity and overhead of these methods low.

#### 4.3.3.1 Fault Detection

In SAREK, in addition to the usual faults handled in any BFT system, the replicas must also face attacks on the predictor component that can cause the PREDICT() function to produce arbitrary results. SAREK therefore makes extra efforts to efficiently prevent the false prediction results from compromising the safety of the system. The detection scenarios can be briefly summarized as follows: (1) Once a replica calls the PREDICT() function in the agreement stage, it sets a timer while waiting for the leader replicas responsible for that request to start ordering. If a timeout is already triggered, but the replica does not have enough sub-requests to enter the execution stage, a fault is detected. (2) If a faulty replica intentionally fabricates an incorrect access to a "*missing*" partition during execution and attempts to cause a re-prediction (see Section 4.3.4), this is also detected as a fault. Table 4.1 classifies the scenarios according to their causes as well as consequences.

**Algorithm 4.3** Detect and resolve cycle

```
1  initialize:
2      blocked_by_temp as a snapshot of the current blocked_by array
3      Node as a data structure defined as ⟨request, instance id⟩
4      seen := NULL // empty linked list indicating every Node seen so far is blocked by its successor
5      ℱ := NULL // empty set to store free instances
6      ℬ := NULL // empty set to store blocked instances

8  detect_and_resolve_cycle(self_id, self_req) do
9      // mark the partition itself also blocked
10     blocked_by_temp[self_id] := self_req
11     // determine blocked partitions
12     for bft_id from 0 to no_of_partitions − 1
13         if blocked_by_temp[bft_id] equals NULL
14             ℱ.add(bft_id)
15         end if
16     end for
17     do
18         for all bft_id not in ℱ
19             req := PartitionSchedules[bft_id].peek()
20             if req.partitions ∩ ℱ not equals ∅
21                 ℱ.add(bft_id)
22             end if
23         end for
24     while ℱ is changed
25     ℬ := {bft_id | bft_id ∉ ℱ}
26     if ℬ equals ∅
27         return FALSE
28     end if
29     // detect cycle
30     bft_id := min{bft_id | bft_id ∈ ℬ}
31     current_node := Node(blocked_by_temp[bft_id], bft_id)
32     seen.append(current_node)
33     while TRUE do
34         current_req := current_node.req
35         next_id := min{bft_id | bft_id ∈ current_req.partitions
36             ∧ blocked_by_temp[bft_id] not equals current_req}
37         next_node := Node(blocked_by_temp[next_id], next_id)
38         if seen.contains(next_node)
39             loop := the truncated part of seen starting at next_node
40             break
41         seen.append(next_node)
42         current_node := next_node
43     end while
44     // resolve cycle
45     req_to_move := arg min_req {bft_id | (req, bft_id) ∈ loop}
46     for all bft_id in req_to_move.partitions
47         move req_to_move to head in PartitionSchedules[bft_id]
48     end for
49     return TRUE
50 end
```

Table 4.1: Causes and consequences of faults in SAREK.

| Faults in SAREK | | |
|---|---|---|
| **Scenario** | **Cause** | **Consequence** |
| **During Agreement** | (1) **Missing sub-requests.** A slow or faulty replica cannot initiate the sub-request ordering in its leader instance(s). | The timeout triggers a view change. |
| | (2) **Additional sub-requests.** A faulty replica manipulates the prediction and fabricates a sub-request. | The faulty replica's behavior does not affect correct replicas and will eventually be suspected. |
| **During Execution** | (3) **Incorrect re-prediction.** A faulty replica can intentionally initiate a false re-prediction process. | A re-prediction process is processed locally by the faulty replica, but this does not affect other replicas and will eventually be suspected. |

#### 4.3.3.2 Fault Correction

Faulty replicas can deny the ordering of a request, or concoct the ordering of an irrelevant request by manipulating a perfect PREDICT() function that always produces accurate predictions. However, this behavior does not affect non-faulty replicas in SAREK, since they all rely on their local PREDICT() functions for correct results. In addition, they will suspect the faulty leaders when they refuse to process a request that eventually leads to a view-change. And the forged ordering proposals are ignored as the non-faulty replicas stick to their own PREDICT() results.

If a faulty replica fabricates a re-prediction (see Section 4.3.4) during request execution to force irrelevant instances to execute the request, SAREK can detect this attempt. Although a re-prediction can be initiated locally, an incorrect re-prediction proposal will never be approved, as it will require the confirmations of $2f+1$ replicas, but will actually be ignored by all non-faulty ones. This prevents a faulty replica from harming the system, and eventually leads to the faulty replica being suspected by others. Details can be found in Section 4.3.4, which is based on a generic approach to handling inaccurate predictions. Once a faulty replica is under suspicion, the non-faulty ones initiate a view-change to remove its leader role from a state partition, requiring changes to the view-change process and checkpoint mechanism.

#### 4.3.3.3 Checkpoints

In single-leader-based BFT systems, checkpoints are generated locally without any prior agreement. This is because in such systems, single-threaded processing and total ordering can guarantee identical ordering of the same set of requests. Therefore, the checkpoint is usually generated by a local counter, and it is necessary to collect distributed checkpoint

certificates to make the checkpoint stable [68, 57, 32]. However, this is not applicable to SAREK for the following reasons: (1) after executing the same number of requests, the states of different replicas may diverge, and (2) due to the existence of cross-border requests, checkpoints should be created at the level of replicas and not partitions. Consequently, synchronization is required before collecting checkpoints.

SAREK performs this synchronization by using a special *create-checkpoint* request which resembles a cross-border request that accesses all partitions. Each create-checkpoint request has its own sequence number and is triggered by $2f + 1$ *pre-checkpoint* messages. Pre-checkpoint messages are sent independently by each BFT instance, as shown in Algorithm 4.4. Each instance maintains a counter indicating how many (sub-)requests (i.e., EXECUTEs, CROSS-BORDER-EXECs, and CROSS-BORDER-SYNCs) it has processed since the last stable checkpoint. After a predefined threshold is reached, the instance broadcasts a pre-checkpoint message with the replica ID and the expected next sequence number. Algorithm 4.5 describes the process of handling the pre-checkpoint message by a replica. The replica records the sequence number of the last stable checkpoint. If the sequence number in the received message is stale or was provided by the same replica with a different BFT instance, the message is immediately discarded (line 6), otherwise it is buffered by the instance. Once the replica has received $2f + 1$ pre-checkpoint messages with the same sequence number, it generates a create-checkpoint request and starts processing like a regular client request accessing all partitions. The execution stage (line 13 of Algorithm 4.2) of the create-checkpoint request looks like this:

1. Create the checkpoint.
2. Each BFT instance resets its counter $c$ to 0, and updates $cp\_next$ to $\max(cp\_next, s+1)$.
3. The replica updates $cp\_stable$ to $s$.

This creates a stable checkpoint consistently across all replicas. To update the state of a slow or recovering replica, a stable checkpoint is acquired from another replica, as well as $f$

**Algorithm 4.4** Send pre-checkpoint at each instance

```
 1 initialize:
 2     interval as checkpoint interval
 3     cp_next := 1
 4     c := 0

 6 while executing at instance bft_id of replica i do
 7     if PartitionSchedules[bft_id].dequeue() is called in Algorithm 4.2
 8         c := c + 1
 9         if c mod interval equals 0
10             broadcast ⟨PRE-CHECKPOINT, i, cp_next⟩
11             cp_next := cp_next + 1
12         end if
13     end if
14 end while
```

matching hashes of the same checkpoint from other replicas to prove that the checkpoint is valid.

#### 4.3.3.4 View Changes

In existing BFT systems, during a view change, the new leader is usually selected deterministically, based on a round-robin strategy. In SAREK, on the other hand, a faulty replica is stripped of its leader role on one partition and assigned to one of the other replicas through a view change. The challenge with SAREK is that after multiple view changes, different partitions are affected, so it is possible for one replica to be responsible for multiple partitions at the same time, which affects the load balancing and parallelism policy. In the worst case, if a single replica is assumed to be the leader for all partitions, the system would degrade to a single-leader-based BFT system. To avoid this scenario and keep the leader roles distributed, we define a *preferred leader* for each partition. This means that after a view change, SAREK will execute the agreement protocol for a predefined number of requests using the new leader and then return to the preferred leader. If the fault that caused the previous view change is temporary, the system will stay with the preferred leader, otherwise another view change will be triggered. To prevent too frequent view changes, we add a switching penalty for the preferred leader after each view change, which increases the waiting time (e.g., by increasing the number of requests to execute) for returning to the preferred leader.

### 4.3.4 Imprecise Prediction Handling

SAREK can use imprecise application knowledge to make predictions if implementing a perfect PREDICT() function is not feasible. In particular, the system is able to address *mispredictions* caused by unforeseen data accesses or internal state changes.

**Algorithm 4.5** Order create-checkpoint at each replica

```
 1 initialize:
 2     cp_stable := 0

 4 upon receiving ⟨PRE-CHECKPOINT, rep_id, s⟩ at replica i do
 5     if s ≤ cp_stable or already received the same message before
 6         discard the message
 7     else if received PRE-CHECKPOINT from 2f + 1 replicas with the same s
 8         for each bft_id in all BFT instances
 9             if i equals leaderOf(bft_id)
10                 start agreement stage of ⟨CREATE-CHECKPOINT, s⟩
11             end if
12         end for
13     end if
14 end
```

#### 4.3.4.1 Re-predictions

To ensure consistency across replicas, SAREK monitors state access by each request during its execution stage. Once the execution instance detects that, a request attempts to access a partition that is not included in the prediction result, indicating a misprediction, the instance immediately suspends execution and prepares a re-prediction. To do so, the execution instance first adds the affected partition to the original request and then sends it to the responsible agreement instance of the local replica as a *re-prediction proposal*. Next, the agreement instance broadcasts the proposal to all other replicas for validation. When a replica has received $2f + 1$ matching re-prediction proposals, it can prove the correctness of the re-prediction attempt, and the leader replica starts ordering the proposal, as shown in Figure 4.6.

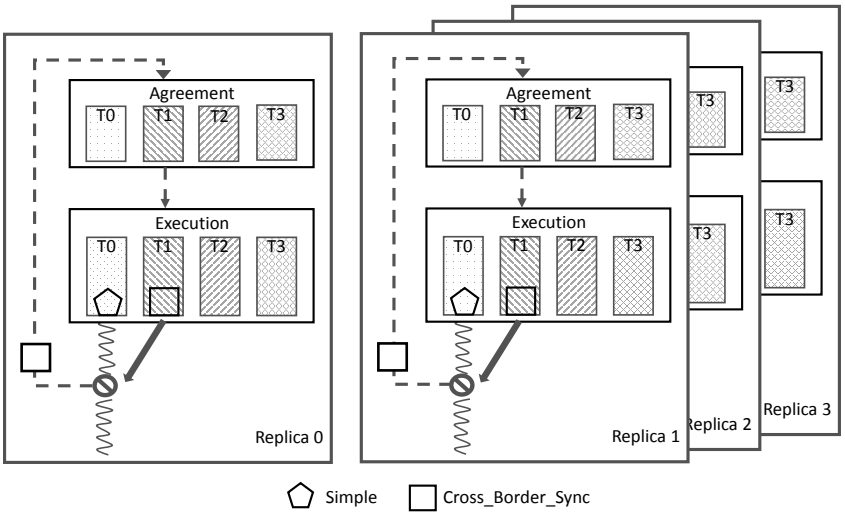



Figure 4.6: Handle inaccurate prediction with re-prediction.

The execution instance $T0$, after detecting a misprediction, sends a proposal for re-prediction to the responsible instance $T1$. The instance $T1$ of one replica then broadcasts the proposal to all other replicas. With $2f + 1$ matching proposals, the leader replica of $T1$ (assumed to be $Replica1$) can safely order the proposal. Ordering the re-prediction proposal ensures that any data access to the re-predicted partition is synchronized and consistent across all replicas. After the commit, the proposal is classified as CROSS-BORDER-SYNC and forwarded to the corresponding instance of the execution stage. The corresponding instance can then notify the halted execution instance when the CROSS-BORDER-SYNC finally reaches the top of its schedule, using the request scheduler as needed to remove possible request cycles caused by already scheduled requests. Upon receiving such notification, the execution instance resumes the halted process and can safely access the affected partition.

The mechanism for handling re-predictions in SAREK guarantees that faulty replicas cannot trigger unnecessary re-predictions. Although a faulty replica is still able to make an

incorrect re-prediction proposal, in the presence of at most $f$ faulty replicas, it will not succeed in finding $2f$ other replicas that make the same proposal. As a result, the incorrect proposal is never confirmed by the agreement protocol.

#### 4.3.4.2 Concurrent Re-predictions

As described in the previous section, after detecting an incorrect prediction, SAREK suspends the execution of the request until the re-prediction proposal for the affected partition is committed. Although handling a single re-prediction with this approach is straightforward, handling concurrent re-predictions requires more attention because without additional measures, concurrent re-predictions occurring in different partitions can easily lead to circular waiting. For example, if a request $R1$ executing in partition $P$ wants to unpredictably access partition $P'$, while at the same time a request $R2$ executing in partition $P'$ wants to unpredictably access partition $P$, both requests will eventually block each other's execution.

To solve this problem, we need to design the partitions to ensure that such cyclic dependencies cannot occur. Following the principle of distributed deadlock prevention [99], we use the prioritization mechanism as well as application-specific knowledge, to satisfy the following requirement: A re-prediction can only access partitions $P_i > P_{max}$, where $P_{max}$ is the partition with the highest number among all partitions included in all previous (re-)prediction results of the same request. By following this rule, the concurrent partition accesses from different requests can be automatically serialized so that no cycles are triggered between multiple re-prediction proposals.

Note that the limitation discussed above only applies to the use cases where mispredictions may actually occur because the PREDICT() function is not completely accurate. Other practical approaches to solve this problem without performing an extensive application analysis are to implement the PREDICT() function conservatively, e.g., by including additional partitions in the initial prediction result.

## 4.4 Implementation and Evaluation

We implemented a prototype of SAREK based on a single-leader BFT protocol. Several evaluations were conducted to compare the performance of this multi-leader system adapted to SAREK with the original system, and to show the advantage of parallel agreement and execution by using state partitioning.

### 4.4.1 System Setup

We use a cluster of four machines to host the replicas, and another dedicated machine to simulate the clients. Each physical machine is equipped with an Intel i7-4770 (quad-core with hyper-threading) CPU and 16 GB of main memory. All machines run the same operating system (Ubuntu Linux Server version 14.04 64-bit) and are connected via switched 1 Gb/s Ethernet.

We build our SAREK prototype based on a PBFT implementation provided by a BFT library written in Java [61] that follows the traditional single-leader design. To run multiple BFT instances in parallel, we instantiate the underlying PBFT protocol engine multiple times in SAREK. This way, the same codebase can be used for all evaluations, and switching between the original system and SAREK is just a matter of configuration. SAREK itself adds only about 900 lines of code to the original codebase, including the implementations of the predictor, the multi-leader agreement handler with executors and the re-prediction mechanism, and increases the codebase by only 1/7.

To evaluate the performance of SAREK, we used two benchmarks: (1) a microbenchmark running a key-value store application, and (2) the YCSB (Yahoo! Cloud Serving Benchmark) [94] benchmark with a customized database server. In what follows, we refer to the original single-leader approach as "baseline (BL)".

### 4.4.2 Microbenchmark Setup

For the microbenchmark measurement, we use a hash-map-based datastore to evaluate the performance (i.e., latency and throughput) and resource requirements (i.e., usage of CPU) of SAREK with different types of workloads. The datastore implements the following functions: (1) it accepts any client request that contains specific keys accessing one or more objects; and (2) it generates a reply and returns it to the client.

The request and reply messages vary in size depending on the operations performed. For a $get()$ operation that returns the data corresponding to a given key, the size of the requests is usually smaller than the size of the replies. In contrast, when creating data in the datastore with a $put()$ operation, the requests are usually larger than the corresponding replies. In addition, the datastore provides a $putall()$ operation that combines multiple updates into one request, which allows testing the handling of cross-border requests in SAREK.

To implement state partitioning in SAREK, we divide the service state into four partitions of approximately equal size. The actual partitioning is done logically with a PREDICT() function that performs a $modulo$ operation on each key to determine the ID of the BFT agreement instance responsible for ordering that request. For all measurements, we deploy up to 200 clients to saturate the system, and compute the average of multiple independent runs as the final result.

### 4.4.3 Microbenchmark Results

For an initial throughput evaluation, we use a combination of requests and replies of different sizes: 50 bytes/500 bytes, 500 bytes/50 bytes and 500 bytes/500 bytes for request/reply, respectively. Therefore, this evaluation is performed to gain insights into the performance of read-heavy ($get()$), write-heavy ($put()$), and mixed workloads. Note that we only consider simple requests in this evaluation; cross-border requests will be investigated later. Figure 4.7 shows that the maximum throughput of the baseline system is less than 20,000 requests per second, while it is about twice as high with SAREK. It also shows that for both systems, a smaller request size apparently leads to a higher throughput, while the reply size has a much smaller impact.

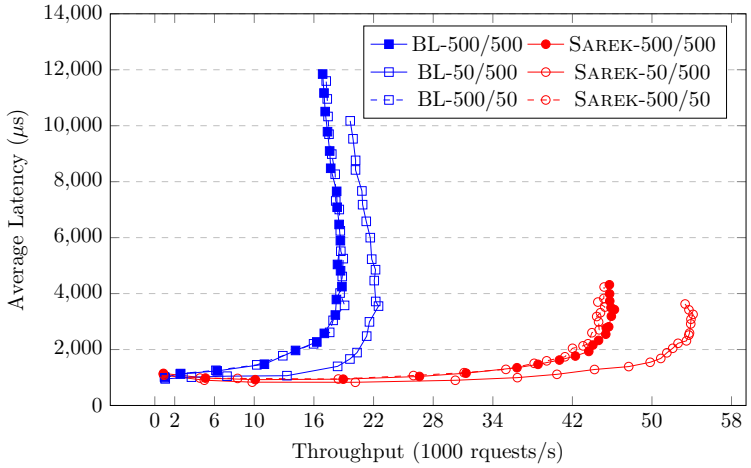



Figure 4.7: Throughput and latency of simple requests.

In our second evaluation, we measure the impact of using request batching, an optimization method that is commonly used in BFT systems to minimize agreement overhead by ordering batches of requests instead of processing each one individually [68]. We set the maximum batch size to 100 and run the evaluation with a request/reply size of 500 bytes/500 bytes. From Figure 4.8 we can observe a significant increase in throughput after using request batching for both systems, while the leading position of SAREK remains the same by almost a factor of two. Therefore, we conclude that request batching can essentially be considered as an orthogonal technique that can be combined with SAREK.

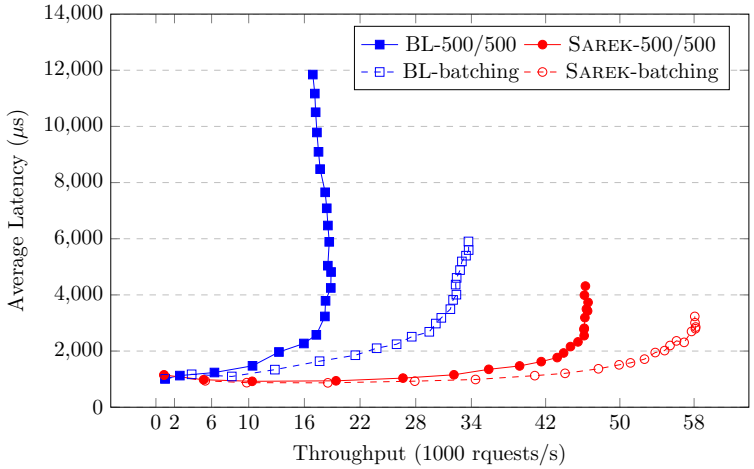



Figure 4.8: Throughput and latency of simple requests with request batching.

In the next evaluation, we measure the impact of increasing the size of request and reply

messages on system throughput. From Figure 4.9, we can see that the throughput of the baseline system is limited because only a single leader limits the processing capability. On the other hand, SAREK can further increase its throughput and outperforms the baseline system by 200%, until the bandwidth limit is reached.

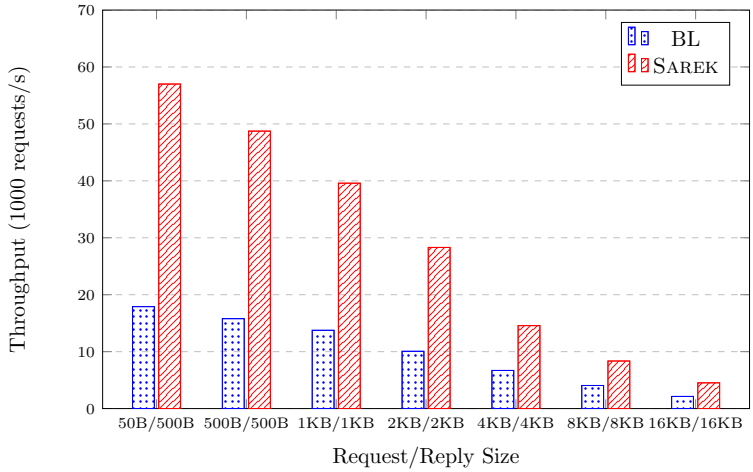



Figure 4.9: Throughput with increased request/reply sizes.

As a key feature of SAREK, the performance of cross-border requests handling is also one of our main interests. Figure 4.10 shows the throughput of both systems when different amounts of cross-border requests are present in the workload. Although the throughput of SAREK decreases by 20% and 30% in the cases of having 10% and 30% cross-border requests, respectively, it is still better than that of the baseline system by at least 50%. Moreover, even at 30% cross-border requests, SAREK is still able to significantly reduce the average latency for each request.

In the next evaluation, we decide to expose SAREK under an extreme condition by increasing the fraction of cross-border request to 100%, and measure performance in three settings: a request accessing data from two partitions, three partitions, and four partitions. Figure 4.11 clearly shows that even with the overhead of processing 100% cross-border requests, SAREK can still outperform the baseline system when the requests access two partitions. For the requests accessing three and four partitions, SAREK's throughput falls behind the baseline system only when there are 90% and 70% cross-border requests, which is due to the overhead of ordering a cross-border request multiple times during the agreement stage. However, if such a hard condition is true in reality, it already indicates that either the majority of requests are highly interdependent and therefore parallelism can hardly be introduced, or the implemented partitioning mechanism needs to be revised.

We also measured the CPU usage of both systems when they have their peak throughput. Figure 4.12 shows the result and proves that the system benefits from SAREK's load balancing feature. If we compare the CPU usage of a replica in SAREK with that of a follower replica in the baseline system, the former consumes about 1.5% more CPU than the latter, since

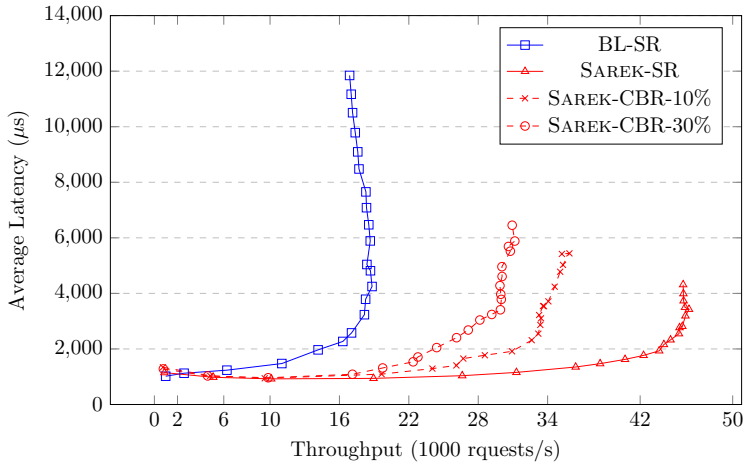



Figure 4.10: Throughput and latency with different amounts of cross-border requests.

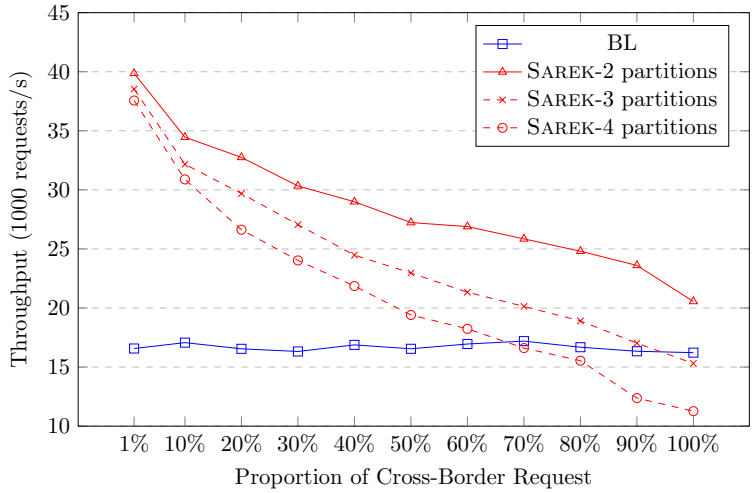



Figure 4.11: Throughput of an increased number of cross-border requests.

the load in SAREK is distributed more evenly across all replicas. Moreover, for the same throughput, a replica in SAREK consumes about 8% less CPU than the leader replica in the baseline, delaying system saturation.

### 4.4.4 YCSB Benchmark Setup

The Yahoo! Cloud Serving Benchmark (YCSB) [94] is a suite of programs for measuring the performance of NoSQL database systems, and is used to further evaluate SAREK. We created a new key-value store application by extending the `DB` class of the YCSB library to

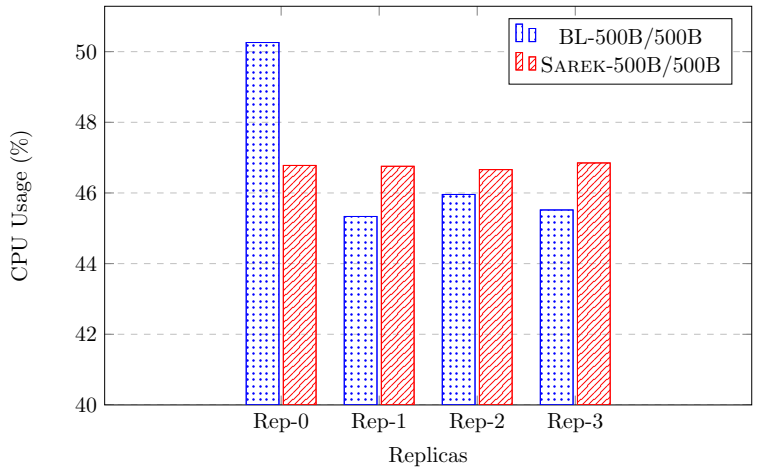



Figure 4.12: CPU usage at peak throughput.

use its client-side benchmarking APIs. This application supports the following operations: *insert*() creates a new record in the datastore, *read*() retrieves an existing record, *update*() modifies an existing record, and *scan*() performs a range scan that queries a specified number of records. For each evaluation, we initialize the datastore with 4,000 records inserted and run the client with one of the predefined YCSB workloads: *read/update* and *scan/update*. On the client side, 60 client instances are created.

The state partitioning mechanism of this application remains simple, since the *insert*(), *read*(), and *update*() operations access only one record, with the exception of *scan*(). Therefore, we partition the entire key space of the datastore into four contiguous segments such that each segment contains a set of consecutive keys. Depending on the density of keys distributed in each partition and the defined scan range, a *scan*() may result in handling cross-border requests that access partitions in a monotonic order, as described in Section 4.3.4.

#### 4.4.4.1 Read/Update Workload

For this workload, a read/update ratio of 50/50 is used to evaluate the ability to handle an update-heavy workload, where each client instance issues 200,000 operations.

#### 4.4.4.2 Scan/Update Workload

For the scan/update workload, the percentage of *scan*() operations is between 5% and 25%, and client traffic is generated at a rate of 100,000 operations per client instance. The maximum scan range is set to 10 (keys) because the key distribution in the entire key space is rather sparse with the 4,000 initial records. In particular, the *scan*() operation provides a range query that starts at a given key and continues to the end of the scan range. Note that since only the starting key is known to the PREDICT() function, the expected result of the prediction is the partition to which the starting key belongs. Depending on the location of

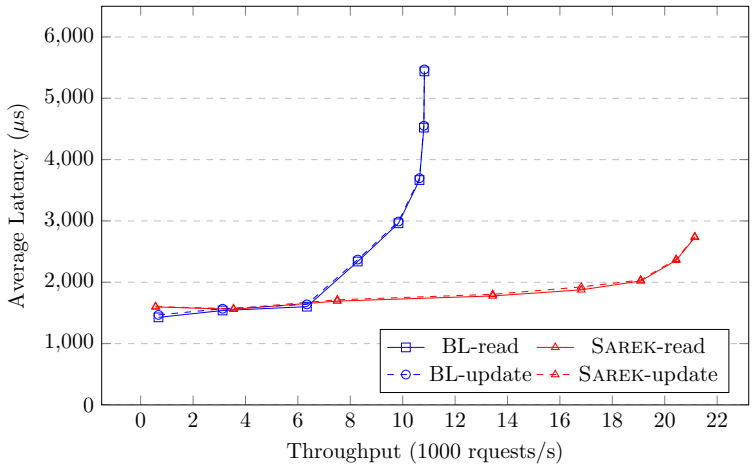



Figure 4.13: Throughput and latency of read/update workloads.

the starting key in this partition, it is quite possible that the *scan*() operation will eventually access multiple partitions. If this is the case, the re-prediction mechanism is activated to handle the case of a misprediction.

### 4.4.5 YCSB Benchmark Results

The results of the YCSB benchmark generally show good performance of SAREK in terms of throughput and latency, and are analyzed separately for each workload.

#### 4.4.5.1 Read/Update Workload

Similar to the microbenchmark, Figure 4.13 shows that SAREK outperforms the baseline system by delivering twice the maximum throughput with significantly lower latency. Note that the average latency of *read*() is not significantly smaller than the latency of *update*() because the agreement protocol is also executed for reads, since we do not include any read optimizations in SAREK. Unlike the baseline system, where request latency increases dramatically at throughputs greater than about 6,000 requests/s, SAREK maintains its low latency up to throughputs close to 20,000 requests/s.

#### 4.4.5.2 Scan/Update Workload

The scan/update workload of the YCSB benchmark leads to a larger reply size and is challenging for SAREK due to the occurrence of cross-border requests and possible misprediction cases. We set the *scan*() operation ratio to 5% and 25%, as shown in Figure 4.14. The baseline system reaches its peak throughput at about 10,000 requests/s, while SAREK reaches a maximum throughput of almost 20,000 requests/s, twice as high. The latency in SAREK

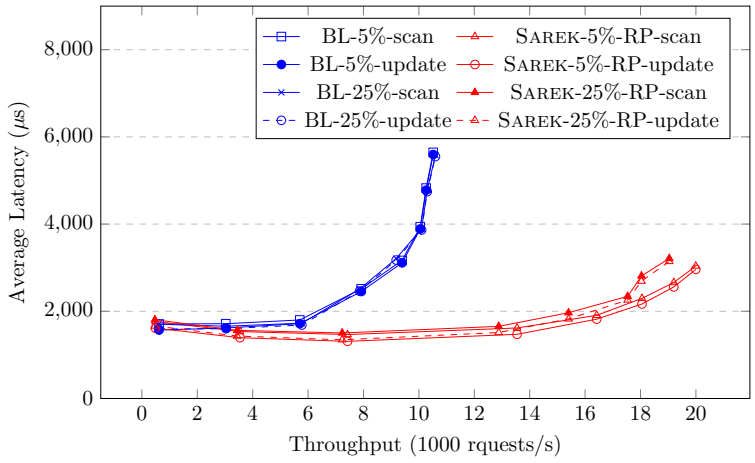



Figure 4.14: Throughput and latency of scan/update workload.

for both $scan()$ and $update()$ operations is significantly lower than the latency of the baseline system for the same operations. As the evaluation result of $scan()$ operation shows, the overhead of handling misprediction cases in SAREK increases the latency only slightly.

## 4.5 Correctness of the Lemmas

In this section we prove the correctness of the lemmas defined in Section 4.3.2.3.

### 4.5.1 Proof of Lemma 1

*Proof.* When an item in the $blocked\_by$ array is changed from $NULL$ to some request, the corresponding instance will wait for notification. The item can only be changed back to $NULL$ by another non-blocked partition (line 18 and 34), and then the corresponding instance is notified. If line 25 has determined any $bft\_id \in \mathscr{B}$, it must wait at the time the snapshot is taken. All other instances that have the potential to change and notify $blocked\_by[bft\_id]$ will also wait at that time. Thus $blocked\_by[bft\_id]$ will not be changed. And the $bft\_id$ instance will wait until a cycle resolution is performed. □

### 4.5.2 Proof of Lemma 2

*Proof.* Each node in $seen$ can be uniquely identified by its partition $bft\_id$. Thus, the number of nodes in $seen$ is not greater than the number of partitions (strictly speaking, not greater than $|\mathscr{B}|$). This means that the while-loop cannot add an infinite number of nodes to $seen$ and will eventually terminate. □

### 4.5.3 Proof of Lemma 3

*Proof.* Whether a request can be successfully executed without cycles does not depend on the runtime race conditions, but only on the result of the ordering. Only nodes that eventually enter a cycle can be included in $seen$. Assume a node $(req, bft\_id)$ is added to $seen$. In all partitions in $req.partitions$, either $req$ is blocking at the head, or another request is blocking at the head. In the former case, any request in front of $req$ in the partition can execute successfully without cycles. In the latter case, the blocking request is the first request getting into a cycle in that partition. These depend only on the partition schedule and are irrelevant to race conditions. And in at least one partition $req$ is not at the head, otherwise $req$ can be executed immediately, violating the Lemma 1. Therefore, line 35 - 37 will find the next node deterministically. □

### 4.5.4 Proof of Theorem 1

First, we have the following corollary directly from lemma 2 and lemma 3:

**Corollary 1.** *Assume two replicas have the same request order in all partitions at a given time, and there is no cycle resolved since then. If they both added the same node to $seen$ during the Algorithm 4.3, then they end up with the same truncated linked list $loop$ (they could start from different nodes, but if we link the last node with the first node in $loop$, they are the same "loop").*

Then we can ensure that if two replicas have detected the same request cycle, they move the same request to the head (line 44) to resolve the cycle. So we have the following lemma:

**Lemma 4.** *Assume $Replica0$ and $Replica1$ have the same request order in all partitions at a given time, and there is no cycle resolved since then. If a request occurs in $Replica0$ in a cycle and is moved to the top and executed by Algorithm 4.3, then it will be moved in $Replica1$ in the same way before it can be executed.*

*Proof.* Since requests are ordered the same in all partitions of both replicas, a request that gets stuck in a cycle in one replica will also get stuck in the other replica. According to Corollary 1, if one replica detects and resolves the cycle, the other replica will do so in the same way. □

Finally, we can prove that if a sequence of cycle resolution procedures involving the same partition occurs in one replica, the same sequence will occur in the other replica. The result is that the relationship between any two requests remains the same in different replicas, ensuring system consistency.

## 4.6 Reducing Cross-Border Requests

In the previous sections, we proposed a parallel ordering framework SAREK that *logically* partitions service state to exploit parallelism for both agreement and execution in BFT systems. SAREK has been evaluated with several benchmarks and the results show that traditional

single-leader BFT protocols [68] can benefit from processing *non-conflicting*, *independent* requests in parallel.

In SAREK, the number of partitions can be equal to the number of replicas in the system. In this way, each replica becomes the leader of a particular partition and specifies an order for requests to access only the objects in that partition. Operations that span multiple partitions, i.e., *cross-border requests*, are also supported by SAREK to enforce deterministic executions. Although using SAREK can increase throughput by a factor of 2, we have also observed that handling cross-border requests in SAREK causes additional synchronization overhead and leads to performance degradation.

As we noted in the previous sections, partitioning of service state must be done properly to reduce the amount of cross-border requests. For example, knowledge of *request dependencies* can be well used to create high-quality state partitions to increase parallelism during request processing. These dependencies are derived from the patterns of object access, that certain objects are accessed together more often, while some others are accessed individually. However, it is almost impossible to obtain accurate and complete knowledge about the state of an application before the application is actually executed. On the other hand, as long as the request dependencies are constantly changing by either creating new state objects or updating object access patterns, a static state partitioning solution can quickly become obsolete and lead to a loss of performance when processing new requests. In addition, a balanced workload on each replica also plays an important role in good performance. Therefore, objects should be distributed as evenly as possible to fully utilize the compute resources of modern multi-core machines. In short, a low cross-border request rate and a balanced workload are crucial for the performance of parallel BFT systems.

In this section, we introduce DYPART, a dynamic state partitioning framework as a complement to SAREK. DYPART first collects the state of the application and partitions it into partitions. It then applies and reconfigures the state partitions at runtime to improve performance. Assume that in the system startup phase, the object access pattern of each client is unknown and therefore no knowledge about the dependencies of the requests can be obtained, leading to the application of the default partitioning. When processing the requests, each replica monitors the state access to find out the dependencies of the requests and maps them to a graph representing the relationships between these state objects. An update of the state partitioning is associated with the checkpoint mechanism to guarantee determinism and keep the overhead as low as possible. Once the checkpoint threshold is reached, in addition to checkpointing, each replica invokes a graph partitioning algorithm to partition the set of nodes of the graph into blocks with minimized weight of intervening edges and possibly similar size. This ensures that the resulting new state partitions maintain a low cross-border request rate as well as fully exploit the potential of parallel request processing.

Since the checkpoint creation process is called periodically, the state partitions are periodically updated with new knowledge about request dependencies in a checkpoint-interval cycle. This enables the reconfiguration feature of DYPART and makes it adaptable to any dynamically changing state access pattern. The fact that the reconfiguration feature does not involve object transfer between partitions and is performed only *logically* also makes it very efficient to use.

We implemented DYPART on the SAREK prototype by appending a module of the dynamic partitioning method to the existing checkpoint mechanism. We evaluated the implementation with microbenchmarks, where requests are generated to simultaneously access multiple objects from a datastore based on a social network dataset [100]. The evaluations show the performance improvement using DYPART compared to the original performance of SAREK.

## 4.7 Related Works

DYPART uses a graph partitioning algorithm to provide dynamic partitioning to a parallelized BFT system to further improve performance and workload balancing. We discuss related works with a reference to parallel computing in state machine replication and applications of graph partitioning.

### 4.7.1 Parallel Computing

P-SMR [42] shows that parallelism can be achieved by mapping non-conflict requests to different multi-cast groups, according to the application-specific semantics. A follow-up work [43] proposed an optimistic mapping approach as well as a roll-back mechanism to solve the inconsistency problem caused by inaccurate mapping assumptions. Compared to these two works, DYPART, being based on SAREK, is able to deal with arbitrary failures without requiring roll-backs to handle mispredictions.

Alchieri et al. present a new way to dynamically reconfigure the degree of parallelism of replicas to handle different workloads [101]. When handling operations with high conflict rate, fewer threads are needed to reduce synchronization overhead, while low conflict rate operations use more threads to maximize performance. Mendizabal et al. propose a high-performance recovery method in parallel state machine replication [102] to avoid the overhead of using dependency detection for consistency guarantee. This method includes two main techniques: (1) speedy recovery, which allows new commands to be processed concurrently with old commands if they are independent; and (2) on-demand recovery, which can recover only a fraction of the state. The dependency between commands is represented as a dependency graph and is executed when parallel execution is efficient considering resource availability. However, these works only consider the crash-stop model, which is different from DYPART. In addition, to accommodate different workloads, DYPART applies state partitioning reconfiguration instead of compute resources, which guarantees that DYPART can always operate under full computational power.

### 4.7.2 Graph Partitioning Application

Many research works have explored the potential use cases of graph partitioning algorithms. We summarize those applications that share a similar concept to DYPART. In [103] Newman shows how graph partitioning algorithms can be used to solve community detection problems, by mapping the common community inference methods to the min-cut graph partitioning problem.

The work of Glantz et al. [104] explores an efficient static mapping from the parallel processes to the processing elements of a parallel system. It creates an application graph to represent the application's computations and their dependencies, and partitions the graph into blocks of equal size. Different algorithms are evaluated to find an efficient mapping from the blocks to the processing elements with minimized communication cost.

## 4.8 The DyPart Extension

In this section, the DyPart design is described in detail and the integrated high-quality graph partitioning algorithm is presented.

### 4.8.1 Design Details

Figure 4.15 shows an overview of the Sarek system with DyPart for state partitioning and illustrates how they interact conceptually. First, a request is handled by the predict() function of the predictor in Sarek, where the predict() function decides how each object should be mapped to a corresponding partition (①). In the case that an object is *undecided* yet for any partition, it is then set to belong to the default partition that has the smallest ID number. After that, the responsible BFT instances order and execute the request by following the rules defined in Sarek.

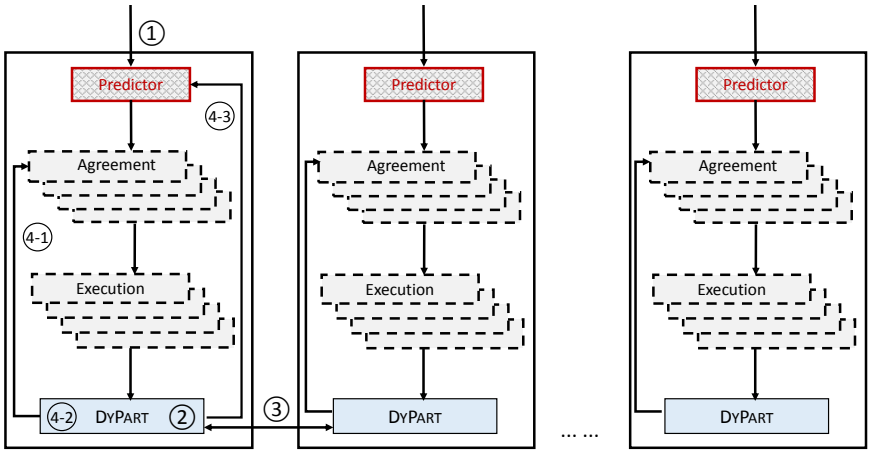



Figure 4.15: Overview of DyPart-backed Sarek system.

The actual process of DyPart is associated with the checkpoint mechanism of Sarek (②). To capture the request dependencies of an application, DyPart logs the co-occurrence of different objects and their frequency by recording each successfully executed request. Meanwhile, after the execution of each request, the checkpoint mechanism checks whether a predefined threshold has been reached. Within a replica, each BFT instance independently maintains a counter that indicates how many requests have been processed since the last sta-

ble checkpoint. When the counter of any BFT instance reaches the threshold, a *pre-checkpoint* message containing the replica's ID and the *expected next* checkpoint sequence number is sent to all other replicas (③). Meanwhile, the next checkpoint sequence number of that instance is incremented by one. Each replica keeps the number of the last stable checkpoint, so any pre-checkpoint message with an outdated number is discarded. Once a replica has collected $2f + 1$ pre-checkpoint messages with the same sequence number, it acknowledges the attempt to create a checkpoint with that sequence number and generates a *create-checkpoint* request. Since the execution speed of BFT instances may vary across replicas, synchronization is required before creating checkpoints. Therefore, the create-checkpoint request is similar to a cross-border request that includes *all* logical partitions. All non-faulty replicas will eventually create this request.

The create-checkpoint request is ordered and executed in the same way that a normal client request (④-①) is handled in SAREK. All involved BFT instances order the request, but only one of them is allowed to execute, while the others do not synchronize until the execution is complete. The execution of this request triggers the creation of the checkpoint and updates the last stable checkpoint number on each replica. Since this request involves all partitions, it acts as a "*barrier*" to ensure that the checkpoint is created in a deterministic manner across all instances on all replicas, despite different execution speeds.

In addition to the checkpoint creation, execution of the create-checkpoint request also causes DYPART to update the state partitioning and eventually the logical partitions by reconfiguring the PREDICT() function. DYPART immediately calls the graph partitioning algorithm, which takes the collected request dependencies as input and generates the new partition knowledge (④-②). The predictor is immediately supplied with this new knowledge to update its PREDICT() function for the incoming requests (④-③). Note that this does not cause any data shift, as it only changes the mapping between the objects and the partitions. Once the predictor is reconfigured, the execution of the create-checkpoint request is complete. It is possible that after the predictor is reconfigured, some requests that have already been ordered are nevertheless predicted using the outdated knowledge, due to the time difference between the triggering of the new state partitioning and the application of the new partitions. If a *misprediction* is triggered in this case, SAREK is able to handle this with its *re-prediction* mechanism.

The optimization can be done by applying the exponential smoothing technique to smooth the continuously updated partition knowledge, since the update of the state partitions is triggered periodically. For example, when computing the new partition knowledge, we can assign a higher smoothing factor to the request dependencies obtained later. In this way, we ensure that the new partition knowledge always reflects the most recent changes to objects and is resilient against unexpected short-term fluctuations.

### 4.8.2 Graph Partitioning Algorithm

The service state is modeled as a graph where the individual objects are the vertices, and the weight of an edge between two objects indicates how often these two objects are accessed together. Therefore, we can transform the state partitioning problem into the graph

partitioning problem, where a graph is cut into smaller pieces and the aggregate weight of edges connecting different pieces should be as small as possible. Similar ideas can be found in high performance computing, e.g., statically distributing work across processors [105]. DYPART features this technique to minimize synchronization between partitions and provide load balancing.

In DYPART, we use the number of requests accessing only one object within the interval of a checkpoint as the *vertex* weight of that object. The *edge* weight, on the other hand, is abstracted from the requests that access multiple objects. For a set of objects accessed together by a single request, each pair of them has an edge with the same weight between them. The weight of the edge increases each time after the associated pair of objects is accessed together. Consequently, if a pair of objects is accessed together more often than other pairs, the weight of the edge is larger. Finally, by examining the accessed objects of all requests, we obtain a graph of objects with different vertex and edge weights. The graph partitioning algorithm is then applied to cut the graph into partitions, minimizing the aggregate weight of edges connecting different partitions. This ensures that the objects with larger edge weights in between are more likely to be assigned to the same partition. The details of the graph partitioning algorithm are described in the following section.

## 4.9 Implementation and Evaluation

We implemented a prototype of DYPART on SAREK's codebase and evaluated its performance with a comparison to SAREK's original, modulo-based partitioning method.

### 4.9.1 System Setup

We use a cluster of four machines to host the replicas (hence we consider $f = 1$ failures) and a dedicated machine to generate client workloads. Each physical machine is equipped with a CPU of Intel Core i7-6700 quad-core 3.4 GHz processor with hyper-threading enabled and 24 GB of main memory. All machines run the same operating system Ubuntu 16.04 64-bit and have OpenJDK 1.8 installed.

We choose Karlsruhe High Quality Partitioning (KaHIP) version 2.0 [106, 107, 108] as the graph partitioning algorithm used in DYPART because of its good performance and easy integration [109]. KaHIP is a multi-level graph partitioning algorithm that recursively contracts a large graph to obtain smaller graphs that reflect the same basic structure as the initial graph. A local improvement algorithm based on max-flow min-cut calculations is used to guarantee the smallest total weight of edges separating the partitions. It is combined with global search strategies to achieve fast graph partitioning. For more details on KaHIP, we refer to its documentations [106, 107, 108].

After KaHIP is compiled, the executable can be invoked as a process within the Java code by DYPART. KaHIP requires the input graph format as used at the 10th DIMACS Implementation Challenge on Graph Clustering and Partitioning [110], i.e., undirected and without self-loops or parallel edges. The input graph $G = (V, E)$ with $n$ vertices and $m$ edges is to be stored in a file of $n+1$ lines with the following format. The first line specifies the number of

vertices and edges and the type of the graph. For example, "80 256 11" represents a graph with 80 vertices and 256 edges, which has both vertex and edge weights. Starting from the second line, the weights are represented in the following structure:

$$c\ v_1\ w_1\ v_2\ w_1 \cdots v_n\ w_n$$

where in a sequence of the vertex ID (omitted in the above structure), $c$ is the vertex weight of the vertex, and $v_1 \cdots v_n$ are the vertices adjacent to the vertex, each of which is connected by a nonzero weighted edge. $w_1 \cdots w_n$ are the corresponding weights of the edges. Note that the weight of the vertex can be 0, which means that this object has never been individually accessed by a request.

DYPART provides an interface to integrate the graph partitioning algorithm into SAREK's codebase as a plug-in service. Each time state partitioning is triggered, DYPART immediately transforms its collected request dependencies into a formatted input graph for KaHIP to perform partitioning. Various options can be added to tune the multi-level graph partitioning program, e.g., to define the number of partitions, to satisfy different quality requirements, or to pursue a balance of edges between partitions as well as vertices. Since we aim for both a low cross-border request rate and a balanced workload for partitions, we choose a partitioning with high quality and balanced vertices and edges to create the output file. DYPART then reads the partition knowledge from the output file into its predictor to update the PREDICT() function.

We evaluate and compare the performance of SAREK with DYPART (noted as *DyPart*) to the original SAREK using a modulo-based partitioning (noted as *baseline*).

### 4.9.2 Microbenchmark Setup

For the microbenchmark evaluations, we built a key-value datastore application based on a hash map, and measured the performance in terms of throughput and latency for both prototypes. The key-value store processes each request for its data access in the datastore, and returns the result. If a request accesses only one object, it is handled by the *put()* operation for writing data; otherwise, it triggers a *putall()* operation for concurrent accesses to multiple data objects.

Since the state partitioning of DYPART is based on the underlying request dependencies of various applications, the input must inherently represent the relationships of the accessed objects. Inputs that lead to random object access are not suitable for evaluation. We consider the social network like datasets as the source for generating cross-border requests that may access more than two objects. More specifically, we choose a co-occurrence network dataset from the Stanford Graph Base [100], which represents the interactions between characters from Victor Hugo's masterpiece "*Les Misérables*".

We abstract the characters' co-occurrences that appear in each scene of each subchapter from the dataset and store them in a list. Each character is considered as a service state object and a vertex of the graph. An undirected edge exists between a pair of characters. The edges of the graph are weighted according to the frequency of co-occurrence of each

pair of characters. Clients randomly select co-occurrence entries from the list to create the requests that access multiple (up to nine) objects.

The original prototype of SAREK relies on a modulo-based partitioning method to obtain four partitions of approximately equal size. For all evaluations, we deploy up to 200 clients to saturate the system. The final result is calculated as the average of the results from multiple runs. Batching is not used for the evaluations because it is an orthogonal approach that affects the results independently.

### 4.9.3 Microbenchmark Results

To find out the efficiency of DYPART and the modulo-based partitioning method of SAREK, we first conduct a measurement to calculate the percentage of cross-border requests generated by these two mechanisms. We go through the request list to see how many partitions are accessed by each request. Figure 4.16 shows that for the modulo-based partitioning method, only 45% of requests access a single partition, compared to over 68% for DYPART, indicating a big advantage of DYPART in reducing the overhead caused by handling cross-border request. For the requests accessing multiple partitions, DYPART is able to reduce the number of requests accessing 2 partitions by 14%, and the number of requests accessing 3 partitions by 6%. Note that the requests accessing 4 partitions, which impose a significant synchronization overhead on the system, can be completely avoided by using DYPART, while this is 3% for the baseline. This shows that DYPART has achieved its goal of keeping a low cross-border request rate.

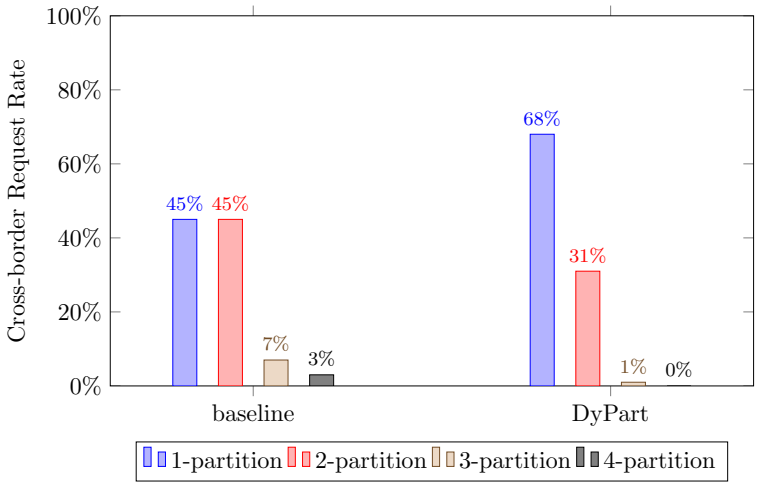



Figure 4.16: Percentage of different request types.

We also analyze the workload balance of the two prototypes and show the results in Figure 4.17. Since the baseline uses the modulo operation to split the state into four partitions of approximately the same size, no request dependency is considered in the partitioning. In fact, in this case, there may be more cross-border requests if the partitions are the same size,

which ends up being more expensive. On the other hand, DYPART prioritizes partitioning quality and balanced vertices and edges, even though some partitions may contain slightly more objects than others.

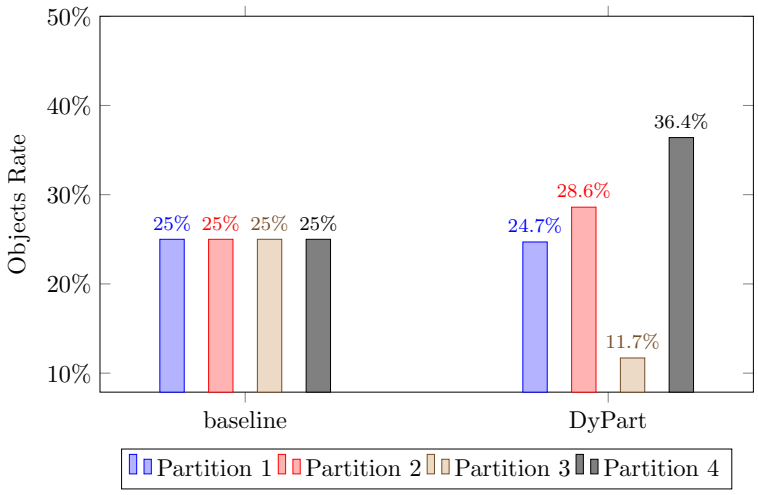



Figure 4.17: Percentage of objects in each partition.

To measure throughput and latency performance, we use two request/reply sizes in our evaluations: 500 bytes/500 bytes and 4 kilobytes/4 kilobytes. On the client machine, up to 200 client instances are created to generate an increasing amount of workload that is tested against the replicas. From Figure 4.18 we can see that when reaching the point where the replicas are saturated (only latency grows, not throughput), using DYPART can achieve much higher throughput than the baseline. For a smaller message size of 500 bytes, a performance improvement of at least 40% can be observed for DYPART; for the 4 kilobytes messages, DYPART outperforms by almost 50%.

Figure 4.19 shows the impact of applying state partitioning and performing re-prediction (see Section 4.8.1) on latency. Measurements are taken every 20 ms, and we show a part of the result after the warm-up period. Using KaHIP to partition the input graph (77 vertices with more than 500 edges) takes 56 ms on average, and the possible resulting re-prediction could also incur a limited overhead. As a result, each time a checkpoint is triggered, a high latency spike can be observed, lasting 40 to 60 ms. By comparing the results of different checkpoint intervals, we learned that increasing the interval can significantly reduce the frequency of such spikes.

## 4.10 Chapter Summary

In this chapter, we introduce SAREK, a parallel ordering framework that relies on multiple leaders to improve the performance of BFT systems. Unlike single-leader approaches, SAREK can distinguish requests according to their data access patterns and specify a partial order

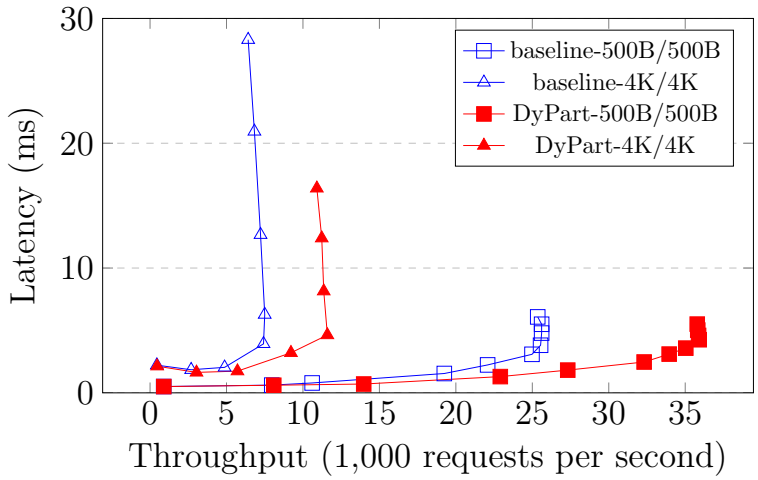



Figure 4.18: Throughput and latency of cross-border requests.

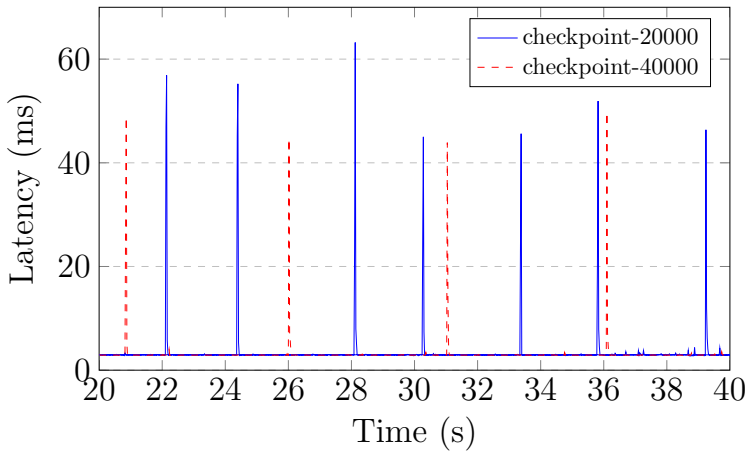



Figure 4.19: Impact of state partitioning and re-prediction on latency.

for the dependent requests instead of a total order. This is achieved by dividing the service state into partitions and managing each of them with an independent BFT agreement instance. In order to distribute the workload associated with ordering, SAREK distributes the leader roles of all BFT agreement instances equally across all replicas under the fault-free condition. Moreover, SAREK provides a way to effectively deal with requests that need to access multiple partitions, especially when the data access patterns are difficult to predict or might even be predicted incorrectly. The evaluation results of the microbenchmarks and the YCSB benchmark show that using SAREK actually increases throughput by a factor of two, even in the case of accessing multiple partitions.

To further improve the efficiency of state partitioning, we also introduce DYPART, a dy-

namic partitioning framework that uses request dependencies to create partitions in SAREK. It collects and analyzes the state of the application with its object access patterns and models the pattern into a graph to represent the relationships of the objects. A high-quality graph partitioning algorithm is integrated into DYPART to partition the state graph into partitions to achieve a low cross-border request rate and balance the workload between partitions. Each replica dynamically updates its predictor component with the new partition knowledge to achieve better performance under this data access pattern. The execution of this process is associated with SAREK's checkpoint mechanism so that it can be consistently applied to all BFT instances across all replicas. We implemented a prototype of DYPART based on SAREK and conducted measurements with microbenchmarks. The results show that DYPART can achieve at least 40% performance improvement over the original modulo-based solution in SAREK by periodically applying partitioning update.

Further optimizations of DYPART can be made to reduce the spikes of high latency (see Section 4.9.3), by decoupling the computation of graph partitioning from the deployment of new partition knowledge. For example, when a checkpoint is triggered, DYPART does not immediately use the results from the current checkpoint interval for reconfiguration. Instead, it applies the partition knowledge from the last checkpoint and completes checkpoint processing directly. In parallel, the partitioning algorithm uses the newly collected request dependencies from the last checkpoint interval to generate new partition knowledge. In this way, the graph partitioning computation does not block the executions at each checkpoint.

# 5

# Transparent Access to BFT Systems

During the last decade, various protocols and architectures have been proposed to increase the practicality of BFT systems. Despite these improvements, the deployment of such systems is still complicated as it requires specific functionality on the client side. This functionality is essential for clients, as they need to connect to multiple replicas and perform majority voting on the received replies to overrule faulty ones. However, deploying custom client-side code in open, heterogeneous systems like we have today can be very cumbersome, especially for established protocols (e.g., HTTP and IMAP) where different client-side implementations coexist, and thus is often not an option.

To make BFT systems *transparent* to legacy clients, in this chapter we introduce TROXY, a system that relocates BFT-specific client-side functionality to the server side. The safety of the relocated functionality is guaranteed by using a trusted subsystem that builds on the hardware protection enabled by Intel Software Guard Extensions (SGX). To further reduce the cost of using BFT systems, TROXY also optimizes the flow of the request process by offering an actively maintained cache for read-heavy workloads. While TROXY is able to perform trusted read operations, it still preserves the consistency guarantees provided by the underlying BFT protocol by transparently switching to traditional request ordering in the event of write contention. A prototype of TROXY was built and evaluated under several conditions: (1) in an experimental local network environment, using TROXY with small messages totally ordered results in a performance loss of at most 43%, revealing the substantial overhead incurred by using the trusted subsystem without any read optimization, while (2) in a simulated wide-area network configured with a real network delay, the results show that using TROXY with read-heavy workloads significantly improves throughput by 130%.

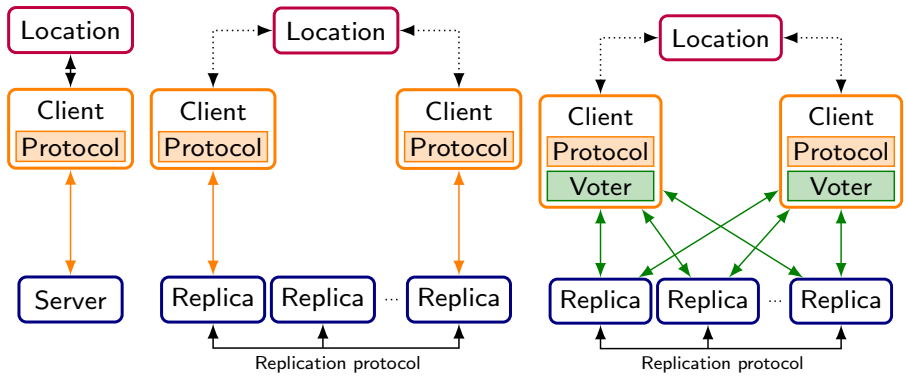
## 5.1 Clients in Distributed Systems

First, we would like to give background on the architecture of distributed systems, in particular how the roles of clients differ between (1) non-fault-tolerant and crash-tolerant systems, which currently represent the vast majority of systems used in production, and (2) Byzantine fault-tolerant (BFT) systems, which have been extensively studied in recent years and are now ready to be deployed in the field. By comparing these two sites, we could gain insight into the impact of moving from existing system architectures (i.e., non-fault-tolerant or crash-tolerant) to BFT replication from a client-implementation perspective. This would help us understand the inherent difficulties in implementing such a move, which largely contribute to the reasons that have prevented many real-world use-case scenarios from migrating to BFT systems to date. Finally, we outline our approach to solving these problems by leveraging trusted hardware to implement a proxy component that resides on the server side so that legacy client implementations can remain unchanged when accessing BFT replication services.

For most network-based services, how a client can access a service generally depends on the server-side implementation. This normally includes the necessary means a client requires, such as hardware resources for computing and software resources such as client-side protocols. In the simplest case, a non-fault-tolerant system contains only a single server, as shown in Figure 5.1a. To access its service, normally a client first queries a location service (e.g., DNS) to obtain the server's address, and then establishes a connection to it directly. The two sides then subsequently interact with each other over this connection based on a specific protocol, such as HTTP for a web service. To ensure the security of client-server communication, many services rely on secure channels such Transport Layer Security (TLS) to handle authentication and encryption/decryption of the exchanged data.

However, if the network services have special requirements for resilience against crashes or scalability of the server, these requirements can only be met if the server-side implementation is replicated to provide crash fault tolerance. In this case, each client usually also maintains only one connection to a single server at a time (see Figure 5.1b). Connections from different clients are typically ensured to be distributed among all available servers to prevent bottlenecks. To reduce the complexity of establishing connections for the client, it is necessary to achieve the distribution of connections in a transparent manner, e.g., by introducing a load balancer [8] in front of the replicated servers, possibly integrated with a location service. Such a mechanism ensures that incoming traffic is distributed across all server replicas capable of handling it so that no single replica is overloaded. It also guarantees that in the event of a replica crash the affected clients are automatically reassigned to other replicas as soon as they attempt to reconnect to the service.

Furthermore, if high availability and resilience against arbitrary faults are required, Byzantine fault-tolerant (BFT) systems offer a solution. Unlike non-replicated or crash-tolerant systems, where a client usually interacts with only a single server replica, BFT systems require a client to communicate with all replicas in the system. In this case, clients not only need to implement the protocol of the service, but also need a *voting component* to safely access the server side [68, 23, 25, 26, 27, 35]. This is because a client in a BFT system cannot trust only




(a) Non-replicated (b) Crash-tolerant replication (c) BFT replication

Figure 5.1: Differences in client perspectives: While the client of a non-replicated or crash-tolerant system usually only interacts with a single server/replica, a BFT client communicates with all replicas in the system.

a single replica, as it could be faulty or even malicious, and therefore may ignore requests or provide incorrect replies. To address this issue, as illustrated in Figure 5.1c, BFT clients do not just contact a single replica, but make connections to all replicas in the system. In this way, they are able to verify the correctness of a result by using the voting component to compare the replies of different replicas. This means that, although the specific communication patterns of clients and replicas may vary between different BFT systems, the commonality of the voting component requires that a BFT client should maintain knowledge of the identity of replicas in order to distinguish their replies. Typically, this information is provided to the client at configuration time. Many BFT systems exploit this knowledge in their implementations to establish a dedicated shared secret between each client and each replica, which is then used to authenticate the messages exchanged. By doing so, they allow a client to verify, among other things, that a received reply actually originated from the presumed replica.

## 5.2 Transparent Access to BFT Replication

While state machine replication-based Byzantine fault tolerance (BFT) was originally considered impractical, the seminal work of Castro and Liskov [68] enabled a stream of research, that improved performance, lowered complexity, and reduced resource usage of BFT [23, 24, 25, 27, 26].

Since Byzantine fault tolerance has been under development for more than a decade, it can already be considered ready for custom deployments today, e.g., it was recently evaluated in the context of permissioned blockchain infrastructures [84]. However, when it comes to the Internet, where various user-facing services are offered in open and heterogeneous environments, BFT still faces a major problem that has been largely overlooked: the *client*

side. As described in Section 5.1, clients in BFT systems must not only implement the protocol of the service, but also use a voter to guarantee the safety of the invoked services (see Figure 5.1c), but this is hardly applicable to most user-facing Internet-based services. This is because these services are dominated by standardized protocols such as HTTP and IMAP, and users of such services typically use different implementations (e.g., different web browsers and email clients) to access these services. In this case, it is infeasible to offer, for example, a BFT-enabled web server, since a client (i.e., a web browser) is expected to contact multiple server replicas and perform majority voting on the received replies to prevent processing of faulty replies. Of course, by extending the HTTP protocol and adding custom software to web browsers [87], one could consider the use of BFT, but this would address only one of many standardized protocols and few client implementations. Therefore, instead of this more or less unrealistic endeavor, we propose to deploy BFT in a *client-transparent* way that is applicable to different types of protocols used in different replicated services.

In this chapter, we present TROXY, a system that achieves client-transparent BFT by moving traditional client-side BFT functionality to the server side. This means that in TROXY, all the essential functions of a client-side BFT library, such as connection handling, request distribution, and majority voting, are moved to the replicas.

To enable this, we need to rely on a trusted subsystem that is resilient to malicious attacks and can only fail by crashing to implement basic message processing, majority voting, and transport encryption for TROXY. At the implementation level, TROXY leverages support for trusted execution as offered by Intel's Software Guard Extensions (SGX) [88, 89]. At the core of SGX is a set of new instructions that allow user-level code to allocate private and secure regions of memory called *enclaves*. By executing application code within enclaves, SGX provides CPU-enhanced application security and protects data and execution within the enclaves from being manipulated by malicious privileged code or even hardware attacks such as memory probes. In this way, TROXY's functionality is guaranteed to be trustworthy even in the presence of Byzantine faults in the surrounding replicas.

Based on trusted execution, TROXY provides clients with a trusted proxy that resides on the server side, allowing them to access the proxy via the original legacy protocol they were using. Once a TROXY instance receives a client request, it forwards the request to the BFT framework, which in turn orders the request, executes it, and forwards the computed replies to the requesting TROXY. As soon as the responsible TROXY instance has received enough replies, it performs a vote on the replies and returns the correct result to the client.

Since the communication of a TROXY instance could be intercepted by the malicious replica host, we ensure that the replica cannot modify messages without being detected: Communications between clients and TROXY instances are protected over secure, encrypted connections, as is common for an increasing number of Internet-based services [90]. In addition, messages exchanged between TROXY instances and replicas are authenticated via shared message certificates, as is common for BFT. While immune to arbitrary or malicious behavior, it is still possible for a TROXY instance to crash or disconnect from its clients, making it unavailable. This case is equivalent to a failed service replica, which is a common failure in commodity infrastructures. To handle such errors, DNS round-robin or load-balancing appliances [111] could be used to allow failover to another TROXY instance.

Since TROXY is specifically designed for user-facing Internet-based services, it also offers tailored support for read-heavy workloads and remote clients, which is achieved by enabling caching for TROXY instances. To avoid reducing the consistency guarantees of state machine replication when using caching, TROXY implements a managed cache for read requests that ensures linearizability [112]. As part of ordering write requests, a quorum of TROXY caches is consulted and the affected data is invalidated. In this way, cached read requests can be answered directly by consulting a quorum of $f + 1$ TROXY instances without ordering them. Otherwise, the requests are ordered using the regular BFT protocol.

## 5.3 Related Works

Traditional BFT state machine replication protocols consist of libraries attached to both the client and server sides [23, 22, 25, 26, 48]. On the client side, the attached library is an essential component mainly responsible for service invocation, message transfer and reply voting, but is hard to integrate into legacy client implementations of user-facing Internet services. TROXY aims to provide a solution to enable transparent and secure connections between the client and the replicated service using trusted computing technology and based on a hybrid fault model [60, 61, 34, 62, 63, 59, 35], where a system is a collection of components with different resilience properties. In this way, the complexity of the replicated fault-tolerant system, in terms of protocol, messages exchanged, and interface is hidden from clients, and legacy client implementations can interact with BFT services without modification. In the following, we discuss related works that use trusted subsystems in BFT systems and provide transparent access to BFT systems.

### 5.3.1 Trusted Subsystem in BFT Systems

TROXY is not the first protocol to explore the use of trusted subsystems in BFT systems. A2M-PBFT [60] is based on a trusted append-only log, which allows it to reduce the number of replicas required from $3f + 1$ to $2f + 1$, compared to traditional protocols. To reduce the complexity of using the trusted log of A2M-PBFT, TrInc [62] is presented as a subsystem that provides trusted counters and can be employed as a less complex replacement for the log. In MinBFT and MinZyzzyva [59], a counter-based trusted subsystem is directly used to make the protocols less complex and more efficient in communication.

The most recent representative of this type of protocols is HYBSTER [35], which also leverages trusted counters to reduce the required replicas to $2f + 1$. The difference between HYBSTER and other hybrid protocols is that, it overcomes the difficulties of the others, such as a time-dependent memory demand, and exhibits a significantly improved performance by introducing consensus-oriented parallelization [27] into the hybrid fault model. In addition to the usage of an FPGA-based trusted subsystem, CheapBFT [34] also exploits passive replication to save compute resources: $f$ out of $2f + 1$ required replicas remain passive and are activated only when faulty behavior is suspected. Similarly, V-PR [63] employs trusted computing technology, called XMHF/TrustVisor [113], to design a fully passively replicated system to tolerate Byzantine failures. By leveraging a trusted subsystem, all these protocols

have a lower complexity in terms of messages exchanged and number of replicas, compared to traditional BFT protocols. Nevertheless, none of the mentioned systems is transparent from the client's point of view.

Avoine et al. [114] present a deterministic fair exchange algorithm running in untrusted hosts with security modules. The untrusted hosts are unable to forge valid protocol messages because the security modules comprise the entire implementation of the consensus protocol. In contrast, the goal of BFT protocols such as HYBSTER (the protocol used by TROXY) is to keep the trusted computing base as small as possible by implementing most parts of the protocol in the untrusted host.

### 5.3.2 Transparent Access to BFT Systems

Prophecy [24] executes a special component between the client and the server, therefore it requires no changes on the client side. As with TROXY, this component must be trusted and acts as a proxy by receiving client requests, collecting replies from replicas, validating a single reply and returning it to the client. However, compared to TROXY, Prophecy requires a large trusted computing base composed of a middlebox, an operating system, and a network stack. Moreover, the consistency model achieved by Prophecy is called delay-once linearizability, which is weaker than the linearizability [112] of traditional BFT protocols: It implies both monotonic read and monotonic write consistency, but no read-after-write consistency, hence the presence of stale reads is possible.

SPARE [115] is transparent to clients by shifting the reply voter component to the server side. SPARE executes replicas inside virtual machines; therefore, it requires a virtualization layer and a hypervisor. It considers a specific fault model: the replicas can exhibit Byzantine behavior while the hypervisor and reply voter can only fail due to crashes. The practicality of SPARE is limited by the large trusted computing base which is composed of a complete hypervisor, a management operating system, and the reply voter.

Thema [116] and BFT-WS [117] extend the classical approach of having a generic client-side library and a server-side library with an additional web-service library. This library collects identical request messages from different replicas, sends the request to a non-replicated web service, and forwards the reply message back to the replicas. Thus, these works address an orthogonal problem and could be combined with TROXY.

## 5.4 The Troxy Approach

To circumvent the previously described problems associated with adding and operating client-side BFT mechanisms, our proposed approach is to introduce a trusted proxy, or TROXY for short, into the system. The TROXY contains the essential client-side functionality of a BFT protocol and acts as a proxy for the client on the server side, allowing legacy client implementations to benefit from Byzantine fault tolerance without requiring any modifications. Since the TROXY is transparent to the client and handles all BFT-related tasks such as reply authentication and voting on the server side, our approach is also suitable for client devices

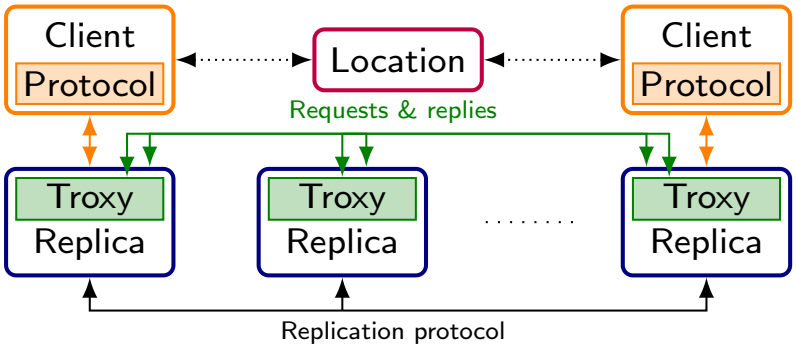



Figure 5.2: Architecture of the TROXY-backed BFT system.

with limited resources and capacity (e.g., mobile phones) as it does not incur any additional network or processor usage on the client.

As shown in Figure 5.2, in a TROXY-backed system, the unmodified client connects only to a single TROXY instance, which then handles communications with replicas in the system for all clients to which it has been connected. If at any time a TROXY instance fails, the affected clients re-establish their connections to the service as they would in a traditional system for reconnection, e.g., via a location service (see Section 5.1), switching to other TROXIES. In contrast to all other replica components, which are untrusted and can fail in arbitrary ways, TROXIES are trusted and are assumed to fail only due to crashes. To justify this trust, we run each TROXY within a trusted subsystem provided by modern processors based on technologies such as Intel SGX [88], which guarantees the integrity of the executed program code. To protect the clients' communication with the service, TROXIES also support the establishment of secure channels using TLS.

In summary, by offering clients with transparent access to BFT systems, our approach greatly facilitates the migration of existing services to Byzantine fault tolerance, as legacy client implementations can be reused without modification or additional resource overhead. At the same time TROXY requires only moderate integration effort into the underlying BFT system at certain extension points.

## 5.5 Design Details

In this section, we present details on the design of a TROXY-backed BFT system in general and on the trusted proxy in particular. For clarity, we defer the discussion of fast-read optimization to Section 5.6.

### 5.5.1 Overview

Figure 5.3 presents an overview of the various components of a TROXY, illustrating how they conceptually interact with each other and with other parts of the system outside the TROXY. When a client issues a request to the service over a secure channel, the TROXY first decodes

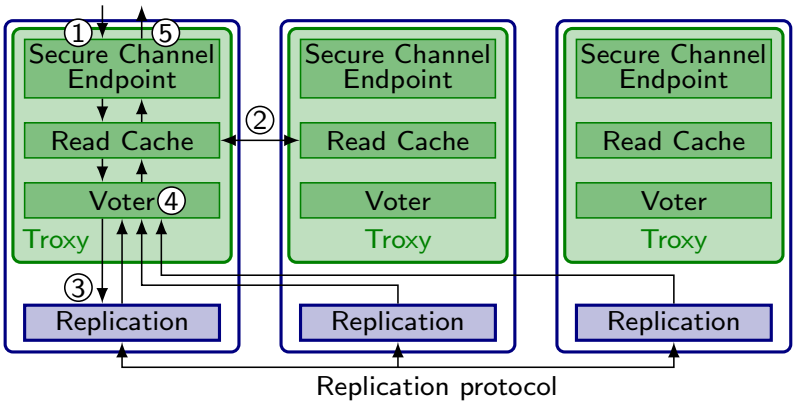



Figure 5.3: Overview of TROXY components and their interactions.

the message (①) after it arrives. For a read request, the TROXY then executes the fast path for reads (②) and, if successful, immediately returns the cached reply (see Section 5.6). For a write request or in the case of a read cache miss, the TROXY forwards the request to its local replication logic to invoke the BFT agreement protocol (③), assuming the role of a BFT client itself. After the BFT protocol receives the request, it distributes the request to the other replicas in the system, ensuring that all correct replicas execute all client requests in the same order. After processing the request, each replica returns the corresponding reply to the replica to which the client is connected, and the voting component of the TROXY then determines the correct result by comparing the replies of different replicas (④). To tolerate $f$ faults, the voter waits until it has received $f + 1$ matching replies from different replicas before returning the result to the client (⑤). This guarantees that at least one of the replies is from a non-faulty replica and is therefore correct. In summary, by acting as a BFT client for the replication protocol, a TROXY already takes care of all the additional responsibilities necessary to access a BFT service, thus freeing the client from the need to perform these tasks itself.

The TROXY approach is based on a hybrid fault model [60, 61, 34, 62, 63, 59, 35], which means that in such a system the components have different resilience properties. It is assumed that all TROXIES in the system either operate correctly or fail by crashing; in particular, this means that once a client receives a result from a TROXY over a secure channel, it can trust that the result is correct. In later sections, we will discuss how we ensure this trustworthiness for TROXIES by minimizing the trusted computing base of a TROXY (see Section 5.5.2) and using Intel SGX (see Section 4.4).

Apart from TROXIES, all other components of replicas and network in the system can fail in arbitrary ways. The number of servers required in a TROXY-backed system to tolerate such Byzantine failures depends on the BFT replication protocol being run between replicas. For example, when using a traditional BFT protocol [48, 57, 23, 28], at least $3f + 1$ replicas are required to tolerate up to $f$ faults. If a replication protocol itself makes use of trusted components [60, 61, 34, 62, 63, 59, 35], the total number of replicas required can

be reduced to $2f + 1$. Since only the TROXY leverages the trusted computing base, replica components located outside the TROXY do not trust each other. Therefore, communication between components of different replicas is handled by exchanging authenticated messages over the network. If a correct component receives a message that it cannot verify, the component simply discards the message.

### 5.5.2 Trusted Computing Base

When relying on a hybrid fault model, it is crucial to keep the trusted components as small as possible [34], because the more complex a component becomes, the more likely it is to fail in an arbitrary way, for example due to possible program errors. Therefore, we minimize the complexity of the TROXY by executing only those tasks that are critical and actually require to be trusted inside the TROXY to justify the trust placed in it. All non-critical tasks are therefore executed outside the TROXY in the untrusted part of the replica. Essentially, this leads to a design where the TROXY is basically a library whose functionality is used by the untrusted part of the replica via function calls.

In terms of client communication, the separation of critical and non-critical tasks means that most of the network-connection handling can be done outside the TROXY. In particular, this includes managing the connected sockets, handling worker threads operating on those sockets, and performing the actual send and receive operations. Overall, there are only three major critical tasks that must be performed by the TROXY in a trusted manner: (1) When a client connects to a replica, the replica's TROXY controls the establishment of the secure channel and then stores the associated session key to prevent the untrusted part of the replica from impersonating the TROXY. (2) When the client sends a request to the server through the established secure channel, the untrusted part of the replica receives the request message. However, the TROXY is the only one able to decrypt the request using the session key. After decrypting the message, the TROXY checks its integrity and creates a BFT-protocol request that includes the client request as a payload. Finally, the TROXY authenticates the created BFT request using the method expected by the underlying BFT replication protocol (e.g., a keyed-hash message authentication code (HMAC) in our implementation) before passing the request to the untrusted part of the replica. In this way, atomically decrypting the client request and creating a corresponding authenticated BFT request ensures that the request cannot be altered by the untrusted part of the replica without being detected by the other replicas. (3) After executing the request, the TROXY collects the replies provided by different replicas, verifies the authenticity of these replies, and then compares them to determine the correct result. Based on this result, the TROXY creates a reply to the client in a final step and encrypts this message with the session key of the client's secure channel. The actual transmission of the reply message takes place outside the TROXY in the untrusted part of the replica. Since the untrusted part does not have access to the session key, it is not possible for a malicious replica to tamper with the reply message without such a change being detected by the client. Although the malicious replica can still discard the message during transmission, this problem can be handled by the fault handling mechanism described below (see Section 5.5.3).

### 5.5.3 Fault Handling

When the client receives a reply message back from a TROXY, it can trust that the reply is correct. In the event of failures, such as when the server hosting the TROXY has crashed, there may be situations where a client does not receive the reply to its request. To handle such scenarios where a TROXY stops working, we take advantage of the fact that clients of user-facing services are typically already equipped with a mechanism to automatically reconnect to the service if their existing connections time out, e.g., by relying on an external location service to help failover to another replica (see Section 5.1). As soon as the client reaches a replica that is not down, it will eventually receive a corresponding reply from the service for that request after retransmitting the request.

Using the same failover mechanism, clients are also able to tolerate fault scenarios in which the untrusted part of a replica that performs the actual send and receive operations on network connections (see Section 5.5.2) fails to deliver the correct reply provided by the TROXY. Depending on the nature of the fault, the client in this case either detects a corrupted channel (if the untrusted part sends data that is not encrypted with the TROXY's session key) or experiences a timeout (if the untrusted part sends no data at all). In either case, the client can solve the problem by reconnecting to the service via a random replica to minimize the likelihood of reconnecting to the same failed replica.

Unlike TROXY, the untrusted part of a replica can fail in arbitrary ways. Apart from the scenarios discussed above, handling such arbitrary failures is mainly the responsibility of the underlying BFT replication protocol, as is the case in traditional BFT systems. The fact that a TROXY, while acting as a BFT client, is co-located with a BFT replica does not affect the protocol's internal fault-handling procedures. Moreover, replay attacks are prevented by the secure channel connecting the client to the TROXY. It is intended that each endpoint never accepts the same piece of encrypted data more than once.

### 5.5.4 Byzantine Fault Tolerance with TROXIES

In the following, we illustrate the steps necessary to migrate an existing user-facing service implemented with a crash-tolerant system to a TROXY-backed BFT system. As an example, we consider a RESTful web service that originally relies on Paxos [92] for fault tolerance and is accessed by a wide spectrum of heterogeneous clients over HTTPS.

The first step in making such a service Byzantine fault-tolerant using our approach is to select a BFT replication protocol and integrate its server-side implementation into the TROXY. This task is greatly facilitated by the fact that the TROXY is essentially a library that needs to be called at a few well-defined places within the replica logic. For example, the TROXY functionality is executed to set up secure channels, safely translate incoming client requests into BFT requests, and determine and encrypt the final replies (see Section 5.5.2). In contrast, the most complex parts of a BFT protocol implementation, such as the ordering and view-change protocols, are left untouched.

The second step is to port the server-side application logic of the web service from the original crash-tolerant protocol to the BFT protocol. For this task, one can usually take advan-

tage of the fact that BFT protocols and crash-tolerant protocols such as Paxos or Raft [118] generally provide comparable interfaces and impose similar requirements on the applications to be integrated, e.g., in terms of execution determinism or the ability to create/apply checkpoints of their state.

In order for the TROXY to communicate with clients, the final step is to make the TROXY aware of the request message format used by the service. In this context, it is not necessary for the TROXY to fully parse and understand incoming requests. Instead, it is sufficient for the TROXY to identify the request boundaries in order to properly store the incoming client request as the payload of the newly created BFT request. And for replies, the TROXY can usually simply extract the payload contained in the verified BFT result and return it to the client. For many communication protocols, including HTTP, identifying message boundaries is straightforward because messages contain information about their own length.

In summary, the steps described above have shown that the overhead caused by such a migration is small if a service is already resilient to crashes. However, since TROXY provides transparent access to BFT systems, even for non-replicated services that previously did not provide any fault tolerance, the changes necessary to integrate Byzantine fault tolerance are limited to the location service (i.e., to make it replication-aware) and to the server-side implementation. However, there is no need to modify the potentially large number of different client implementations.

## 5.6 Fast-Read Cache

The TROXY features a *managed fast-read cache* that not only validates cache entries when processing regular read requests, but also removes entries from the cache when a write request is about to stale the associated cache data. By invalidating cache entries during the processing of write requests and before their effects are emitted to the clients, a TROXY-backed BFT system is able to maintain the consistency guarantees provided by the underlying BFT protocol. In the following, we present details of the fast path of TROXY for read requests using the example of a BFT system based on a hybrid fault model and therefore able to tolerate $f$ faults with $2f+1$ replicas, as is the case with our prototype implementation (see Section 5.7.2).

### 5.6.1 Protocol

Consistent with previous research works [23, 24, 26], our fast-read optimization assumes that read and write requests can be distinguished before they are executed and that it can be determined which part of the state a request should access or modify. The described functionality is executed within a TROXY instance and is therefore trusted, except for functions provided by the surrounding untrusted parts of the replica.

After distinguishing it as a write request, the processing of this request is used by our fast-read cache to remove an outdated entry from the cache before the effects of the write are visible to any client, i.e., before the reply message is returned to its client. To ensure this, we make two important changes to introduce the cache: (1) We modify the voter of

the TROXY to consider another replica's reply only if the reply is authenticated by the other replica's TROXY. As a consequence, this requirement forces a replica to pass the reply to its local TROXY to have an impact on the final result for that reply, allowing the TROXY to learn of a write and subsequently invalidate the associated cache entry that is stale due to the write. To authenticate the local reply, the TROXY computes an HMAC based on a shared secret known to all TROXIES, and an identifier specific to each TROXY instance. (2) We extend the reply message provided by each local replica to include not only the result of the application, but also (a hash of) the original request, so that a TROXY can identify which cache entry to invalidate. As before, a TROXY returns a result to the client only after it has received $f+1$ *matching replies* (which now include the request) from different replicas. In terms of fast-read cache, this means that if a reply to a write request reaches this point, it is assured that a majority of the replicas in the system have invalidated the associated cache entry.

As shown in Algorithm 5.1, when a TROXY receives a read request from a connected client, it first calls *check_cache*, which takes the request provided by the client as input to determine whether the fast-read cache can be applied. Next, it checks if the cache contains data that can be used to answer the request. If no data can be found, the request is ordered and executed like any other regular request. Otherwise, a set of $f$ remote TROXIES is randomly selected and queried by calling *get_remote_cache_entry(r,req)*. This function generates an authenticated message for the replica *r* to query its TROXY about the currently processed request *req*, which is passed to the untrusted replica code for transmission. On the remote side, the receiving TROXY instances validate the message and then check whether the requested data is in the cache (see line 19, Algorithm 5.1). Then they authenticate both the request and the associated reply, and return them to the requesting TROXY. Next, the local request and reply pair is validated if all $f$ remote pairs match the local data. If so, the reply is returned to the client and a successful cache lookup has been performed. In the case of a mismatch, which may be the result of concurrent writes or actions by malicious replicas (e.g., reloading an outdated reply), the fast read fails and the read request moves to the regular ordering phase.

Note that more aggressive use of hashes can reduce the amount of data exchanged. In addition, timeouts can be used to detect unresponsive replicas.

### 5.6.2 Consistency and Resilience to Performance Attacks

In the context of the implemented prototype, we considered a system based on a hybrid fault model that requires only $2f+1$ replicas while still offering strong consistency. To realize such a system design, we need to set the target of TROXY and its fast-read cache in a way that preserves the guarantees of the underlying protocol. This is achieved by immutably entangling the maintenance of fast-read cache with the execution of the underlying protocol, so that an attacker cannot distract replicas and TROXIES to make conflicting statements. With a total number of $2f+1$ replicas in the hybrid fault model, a quorum of $f+1$ replicas is required to complete a write operation in order to provide authenticated replies. Since authentication of replies can only be done by TROXY within the trusted subsystem, these $f+1$ replicas must

**Algorithm 5.1** Cache lookup when processing read requests

```
 1 // Cache lookup in case of voting Troxy instance
 2 upon call check_cache(req) such that req is READ do
 3     reply := cache.get(id(req))
 4   if reply is not NULL // request is cached
 5       replicas := choose_f_replicas() // select f remote caches
 6       rc := ∅ // set of remote cached replies
 7       // collect cache entries of f remote replicas
 8       ∀r ∈ replicas, rc.add(get_remote_cache_entry(r,req))
 9       // remote caches match local cache
10       if ∀(r_req, u_rep) ∈ rc, (id(r_req), u_rep) = (id(req), reply)
11           return reply // fast read succeed
12       else return null // mismatch amongst caches
13       end if
14   else return null // cache miss
15   end if

17 // Cache lookup in case of remote Troxy instance
18 upon call get_local_cache_entry(req) do
19   reply := cache.get(id(req))
20   return (req,reply)
```

then have cleared the associated entry in their fast-read cache while processing the write request, before the validated reply becomes visible to any client. Meanwhile, a successful fast-read operation also needs $f + 1$ identical entries from the fast-read cache, which means that at least $f + 1$ replicas must still have a matching entry in their cache. This is not possible because both quorums are overlapped by a replica and its TROXY, which is responsible for providing the necessary response to either side, runs within a trusted subsystem. This ensures that a successful fast read reflects the state of the latest write. One possibility for an attacker would be to reset the trusted subsystem by restarting it, which would lead to the cache simply losing all of its state. In this case, however, the consistency guarantee is not violated because the fast-read requests are returned unanswered, resulting in regular execution of the underlying protocol for the requests. Furthermore, forwarding a reply due to a write request always results in a cache *invalidation* instead of a cache update. This is necessary because the local TROXY doing the forwarding can only confirm the origin of the reply, not its correctness. A cache invalidation ensures that a faulty replica cannot pollute the cache.

This leads to the question of whether a faulty replica can negatively affect system performance beyond what it can do in a traditional system. As in classical BFT systems such as PBFT [68], which feature a read optimization that requires $2f + 1$ replicas, a client can only use the result if all replies match. Faulty replicas can thus often prevent successful read optimization by constantly returning non-matching results. In the case of TROXY, we are in a similar situation as we query $f$ random TROXIES for their cache entries. However, we additionally measure the cache miss rate within TROXY. When the miss rate reaches a configurable system constant, fast-read optimization is avoided in favor of a traditional protocol run to process the request. In the case of write contention, where a concurrent write oc-

curs during a fast-read operation that results in many cache misses due to conflicts in state, this can also be addressed by avoiding fast-read optimization, as shown in the evaluation result (see Section 5.8.3.3).

## 5.7 Troxy Implementation

In this section we present our prototype of a TROXY-backed system, with details on the SGX-based TROXY implementation as well as its integration with the BFT protocol HYBSTER [35].

### 5.7.1 Implementation Details

Our implementation of the TROXY component is written in C/C++ and relies on Intel's Software Guard Extensions (SGX) [88] and its SDK [119] to achieve isolation between the trusted and untrusted parts of a replica. In the trusted part, the TROXY runs inside a trusted execution environment provided by SGX, called an *enclave*, which is protected by the CPU via transparent memory encryption and integrity checking. Entering and exiting an enclave is possible only through an enclave interface, which defines the entry points and the maximum number of concurrent threads allowed at any time within the enclave. An enclave call (*ecall*) is needed for calling enclave functions from the untrusted environment, while an outside call (*ocall*) is explicitly used for calling from an enclave to the untrusted environment. An *ecall* initiates the following operations: Executing a translation lookaside buffer (TLB) flush, switching to a trusted stack that is inside the enclave, copying the parameters from untrusted memory and calling the specified trusted function. Similarly, an *ocall* results in a TLB flush, switching back to the untrusted stack, moving the parameters out of trusted memory, and performing an exit of the enclave. Since these operations can cause high overhead [120], it is best practice to minimize enclave transitions to minimize performance loss when using SGX.

TROXY implements *ecalls* for data transfer between enclaves and the untrusted environment, and for data processing within enclaves. To keep the interface small, TROXY defines only 16 *ecalls* and no *ocalls* under a security-aware programming model. More specifically, these *ecalls* have been manually verified and are hardened to prevent possible attacks such as Iago attacks [121] or time-of-check-to-time-of-use attacks [122]. For example, data transfer between the untrusted environment and enclaves requires additional copies of message buffers. A read buffer is always copied directly into the enclave to avoid time-of-check-to-time-of-use attacks; in contrast, the copy of a write buffer can be done outside the enclave to achieve better performance.

To enforce the validity of enclaves, Intel provides a remote attestation service [88]. In short, a hash of the enclave's memory pages is securely computed and sent to the remote attestation service so that the user can obtain proof that the enclave was correctly initialized. Once the enclave has been correctly attested, it is possible to provision it. All cryptographic keys and secrets, such as the private key used by the TROXY to initialize secure connections with clients, can be securely sent to the enclave during the provisioning phase and cannot be

retrieved by other processes or the operating system, so they are protected from malicious replicas.

The enclave's code and data are stored in the Enclave Page Cache (EPC), a special region of memory that is protected from untrusted access. In the current implementation of Intel SGX, this memory area has a maximum size between 128MB and 256MB. Accessing memory that exceeds the size of the EPC results in costly paging, as pages must be encrypted and integrity protected before being paged out to main memory. Since this process causes a high overhead on performance [123], we limit the memory allocations to keep the memory footprint as small as possible. Also, the TROXY can store data encrypted outside the enclave to avoid extra *ocalls* and paging [124]. When the encrypted data needs to be accessed, it is read directly from the untrusted memory and validated by comparing it to a hash securely stored in the TROXY.

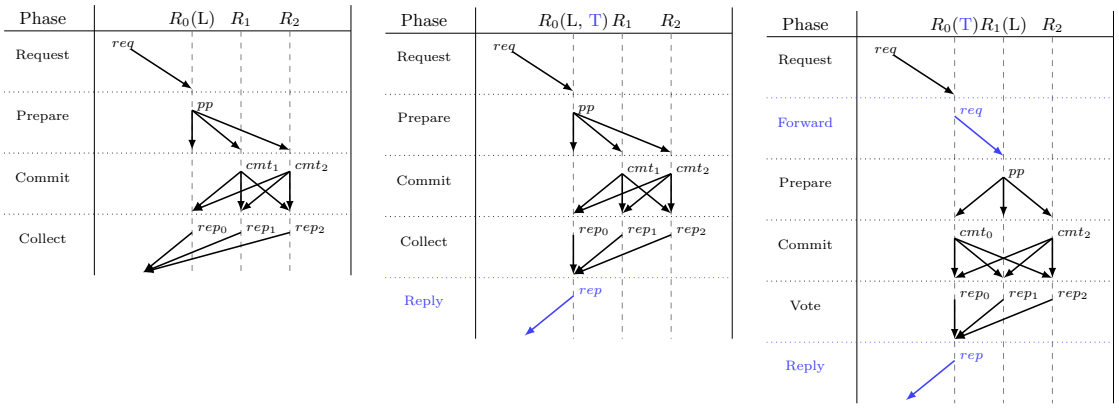
Finally, the TROXY provides bidirectional TLS authentication for all messages exchanged between clients and replicas. To this end, the TROXY uses the TaLoS [125] library, which also leverages Intel SGX technology. TaLoS provides a TLS interface to existing applications and runs the TLS logic securely within an enclave. Note that we run it fully encapsulated: There are neither *ecalls* nor *ocalls* between the TaLoS library and the untrusted environment.

### 5.7.2 TROXY-backed HYBSTER

To provide fault tolerance, our prototype implementation relies on HYBSTER [35], a BFT replication protocol based on a hybrid fault model and therefore only requires $2f + 1$ replicas to tolerate $f$ Byzantine faults. HYBSTER is implemented in Java and uses Intel SGX to realize a trusted subsystem for message authentication. The highlight of HYBSTER is that it achieves high performance through parallelization, with performance scaling well with the number of network interface controllers (NICs) and CPU cores. The trusted subsystem of HYBSTER is also used by TROXY for trusted authentication on internally exchanged messages during the ordering phase. In our implementation, the interaction between the protocol running in the untrusted part of the replica and the SGX enclave is handled by the Java Native Interface (JNI).

HYBSTER is a leader-based BFT protocol: a special node, as a leader, is responsible for proposing an order for the requests received from the clients. Figure 5.4 shows the message flow in the resulting TROXY-backed HYBSTER system. Compared to the original HYBSTER (see Figure 5.4a), the introduction of TROXY adds an additional message delay for a client connected to the leader replica of HYBSTER (see Figure 5.4b). In this additional phase, the corresponding TROXY collects and compares the replies to determine the correct result for the client's request. For clients connected to servers hosting HYBSTER's follower nodes, an additional phase is required for submitting the request to the leader, since only the leader is able to initiate the agreement process for requests (see Figure 5.4c). Note that in an environment where replicas of a system are hosted in different fault domains within the same data center (e.g., different racks with independent power and network supplies [74]), forwarding messages in this additional phase has little impact on the overall latency for the client.

Figure 5.4 also illustrates another important difference between traditional BFT systems




(a) Message flow in HYBSTER.

(b) Message flow in Troxy-backed HYBSTER when the client is connected to the leader replica.

(c) Message flow in Troxy-backed HYBSTER when the client is connected to a follower replica.

Figure 5.4: Comparison of message flows in HYBSTER and TROXY-backed HYBSTER.

and a TROXY-backed BFT system: Since the TROXY performs the replies matching on the server side, the client receives only a single reply per request, rather than multiple replies. In practice, this approach has several key advantages. First, in a typical environment where clients are connected to the service over a wide-area network, sending less data over long distances is especially beneficial for clients with low-bandwidth. Second, in times when wide-area networks are unstable, the response time of the service is reduced by sending fewer replies. This is because, compared to traditional BFT systems, the latency experienced by the client in a TROXY-backed system no longer depends on the arrival of the $f+1$ slowest (for a normal request) or $2f+1$ slowest (for read optimization) matching replies. Third, and most importantly, the BFT replication system becomes transparent to clients.

## 5.8 Troxy Evaluation

In this section, we evaluate the performance of a TROXY-backed HYBSTER system compared to the original HYBSTER using both a microbenchmark and an HTTP service. The results show that: (1) For ordered messages with small payloads in a local network, using TROXY causes an overhead of at most 43% due to its extra communication steps (see Figure 5.4) and transitions to the trusted execution environment. (2) For larger messages with network delay (simulating a wide-area network), using TROXY improves performance by at most 70%. (3) For read-heavy workloads with network delay, fast-read cache optimization improves throughput by 130% even when there are competing write requests. (4) When considering an HTTP service with network delay, the TROXY can almost hide the replication cost, allowing clients to observe similar latency to a non-replicated service.

### 5.8.1 System Setup

Measurements are conducted on a cluster of five identical machines connected via four 1 Gbps Ethernet NICs. Each machine is equipped with an SGX-capable Intel Core i7-6700 quad-core processor running at 3.4 GHz with hyper-threading enabled, and 24 GB of main memory. Three machines are dedicated to replicas (hence we consider $f = 1$ faults), while the remaining two run as clients. All machines run 64-bit Ubuntu 16.04 with a Linux kernel 4.4.0, OpenJDK 1.8, and the Intel SGX SDK v1.9. We compare the performance of our TROXY-backed HYBSTER variant (TROXY for short) with the original HYBSTER protocol, noted as BL (for *baseline*).

### 5.8.2 Security Analysis

In this section we analyze the security of TROXY.

#### Performance attacks

A malicious replica could try to return stale cache entries in the case of fast-read cache optimization. As a result, the fast read attempt would fail and slow down the protocol. As described in Section 5.6.2, TROXY selects $f$ random replicas to reply to a fast-read query and monitors the cache miss rate to address such attacks.

#### Side-channel attacks

We consider side-channel attacks out of the scope of this thesis. However, TROXY can implement existing techniques to limit side-channel attacks within an SGX enclave [126, 127, 128].

#### Bypassing TROXY

A malicious replica could bypass TROXY to break the safety of the system by communicating directly with the clients. To prevent this attack, secure connections using the TLS protocol are initiated by the clients and the TROXY. The session keys of the connections are stored securely in the TROXY, therefore the malicious replica cannot forge correct messages.

#### Interface attacks

A malicious replica could attack the enclave interfaces in order to gain access to the secrets stored in the TROXY. As discussed in Section 4.4, the enclave interfaces have been manually verified and are hardened to prevent such attacks.

#### Denial-of-Service and flooding

A malicious replica might decide to perform a denial-of-service (DoS) attack by not executing the TROXY or following its protocol, or conversely, flooding the correct replicas or clients with

invalid messages. In all these cases, the goal of the malicious replica is to render the system unusable. To prevent such attacks, TROXY can leverage existing techniques [37, 129] such as regular view change or limiting the traffic of attacked replicas.

### 5.8.3 Microbenchmark

We created a microbenchmark to evaluate the full capacity of TROXY and to investigate the overhead of (1) moving the traditional client-side library to the server side and (2) using the trusted subsystem to protect the TROXY. A configured number of clients are created to continuously issue asynchronous requests and measure the average throughput and latency for 60 seconds. The final results are the averages of three runs. Batching is not used as it is an orthogonal approach that has an independent impact on the results.

Secure socket connections are used for client-to-replica communication for both baseline and TROXY, while plain sockets continue to be used for replica-to-replica communication, as well as HMACs for message authentication. Clients connect only to the leader in the baseline system, while TROXY allows connections to any replica. We created a simple service that accepts requests and generates a reply message of configurable size. This service distinguishes read and write requests by their operation types. We conducted experiments in three different scenarios, where (1) all requests are totally ordered; (2) read optimizations are applied to handle read-only requests, and (3) concurrent write requests lead to conflicting read results, resulting in the traditional ordering of conflicting read requests.

In addition to the local network configuration, we added rules to the Linux kernel component netem, which is used to test and simulate the latency conditions of a wide-area network. As a result, a delay of $100 \pm 20$ ms (in a normal distribution model) is added to the client machine NICs. We consider this to be the typical usage scenario of TROXY, i.e., data center hosted services accessed by remote legacy clients.

#### 5.8.3.1 Totally Ordered Requests

In this scenario, we consider write requests of different sizes: 256 B, 1 KB, 4 KB and 8 KB. The size of a reply message is always 10 B. Two implementations of TROXY in C/C++ are compared to the baseline: *ctroxy* runs in the untrusted environment without SGX, showing the impact of using JNI; while *etroxy* runs inside an enclave, adding the overhead of using the trusted subsystem.

Figure 5.5 shows the measurement result of processing write requests in the local network. For a small request payload size (256 B), etroxy shows a performance loss of about 43% due to the transitions between the trusted and untrusted environments and the extra steps in processing ordered requests (see Figure 5.4). More precisely, considering the performance of ctroxy (without SGX), half of the performance loss in etroxy is caused by the use of the trusted subsystem. As the payload size increases, ctroxy and etroxy start to performance similarly and etroxy reaches the baseline at 8 KB. This is due to the fact that authenticating messages with large payloads is faster in C/C++ than in Java.

We also measured performance with simulated network delay between clients and replicas. As illustrated in Figure 5.6, having the reply voter at the server side brings a big ad-

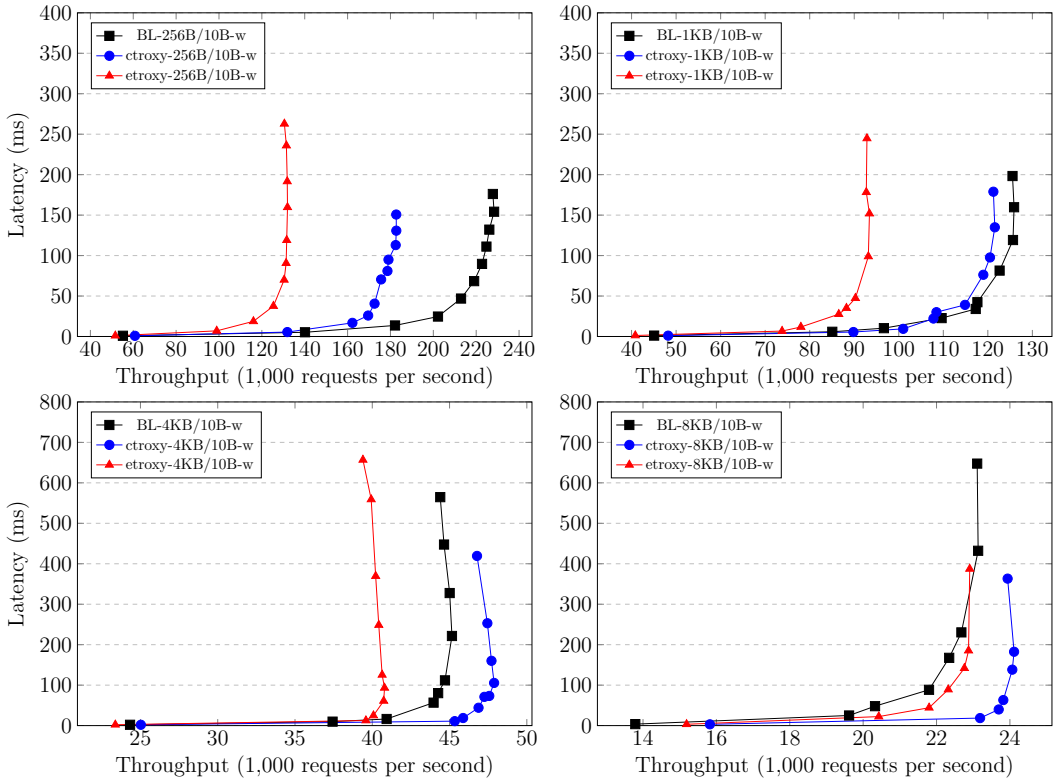



Figure 5.5: Totally ordered write requests in the local network.

vantage to TROXY. In this case, clients only have to wait for a single reply message for each request, instead of $f + 1$ replies affected by the delay. We can also observe that this advantage applies to requests of different payload sizes and leads to a performance gain of up to 60%.

#### 5.8.3.2 Read Optimizations

We also measured the performance of fast-read cache with read-only requests of different payload sizes: 10 B / 256 B, 10 B / 1 KB, 10 B / 4 KB and 10 B / 8 KB for request and reply messages, respectively. The baseline system implements a PBFT-like read optimization approach [68], where read requests are forwarded directly to followers for execution without being ordered. For read-only workloads, this approach can be very effective since there are no concurrent state transitions that create conflicts in the read results.

Figure 5.7 shows the results of handling read-only requests in the local network. Usually, the read requests are quite small and their replies could be relatively large. On the one hand, for small requests (10 B), fast message authentication cannot compensate for the overhead caused by the reply voter on the server side, so an overhead of up to 115% can be observed for 256 B replies. On the other hand, as the size of the reply increases, the effect of fast

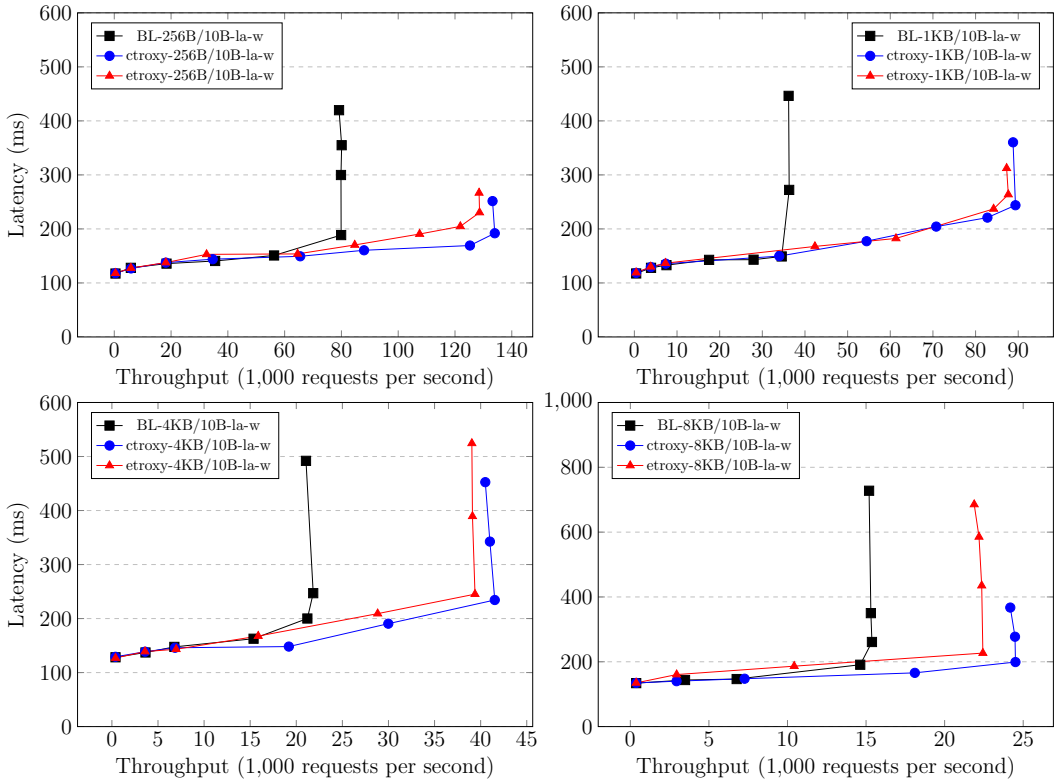



Figure 5.6: Totally ordered write requests with a network latency of $100 \pm 20$ ms.

authentication becomes more pronounced. Therefore, with 4 KB replies etroxy can already outperform the baseline, and with 8 KB we can even observe a throughput improvement of 30%.

The result of the measurement with network delay is shown in Figure 5.8. Although executing reply voting on the server side increases TROXY's overhead, the additional network delay shows less impact on TROXY's performance. Compared to the baseline, with 256 B replies, etroxy only has a 33% performance loss with the network delay and 115% without the delay. Furthermore, since the fast-read cache only needs to transfer the hash of the reply between replicas for a fast-read operation, rather than a full reply, this further reduces the authentication and transmission cost. When the reply size is above 1 KB, etroxy outperforms the baseline by at least 15%.

#### 5.8.3.3 Concurrency Handling

In this scenario, 1% of write requests are generated between reads, to introduce concurrent state transitions during fast-read operations. Due to the different read optimization approaches, the 1% write workload results in different read conflict rates for the baseline and TROXY (only etroxy is evaluated in this scenario). For the baseline, nearly 50% of reads yield

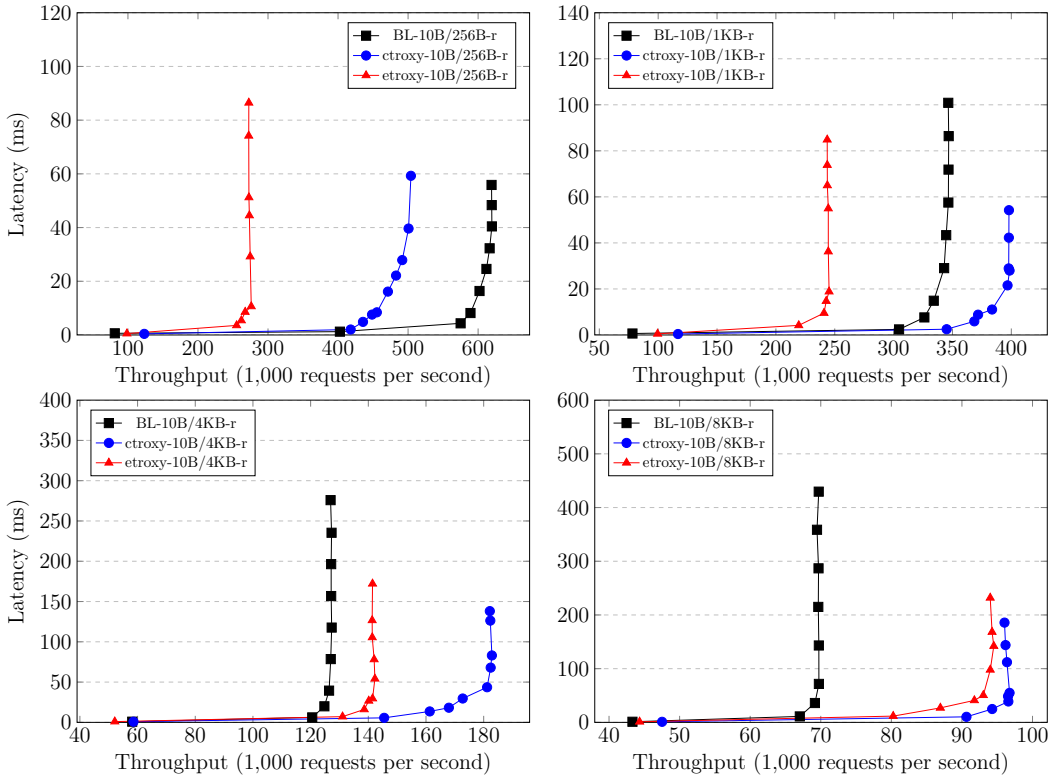



Figure 5.7: Read-only requests in the local network.

conflicting results and fail the read optimization attempt, requiring them to be processed a second time and go through regular protocol execution, adding significant overhead to the system. As with TROXY, fast-read cache behaves conservatively: it uses write requests to invalidate existing cache entries, so that subsequent read requests are directly ordered to avoid conflicts. In this way, TROXY can reduce the conflict rate to 14% as we observed during evaluation.

We also conducted a measurement where no read optimization is applied, so all read requests are ordered to provide a reference throughput for each system for comparison. Figure 5.9 illustrates that the overhead of 50% read conflict contributes to the significant performance loss of the baseline, resulting in read optimization achieving only half of the reference throughput. For TROXY, the 14% read conflict also reduces the performance to a point where it is slightly below the reference throughput. Therefore, we have further optimized the conflict rate monitoring approach within TROXY to ensure that once the conflict rate exceeds a certain threshold, TROXY automatically switches to total-order mode, where all requests are ordered (see Section 5.6.2). This threshold can be learned by sampling the system to determine at what conflict rate the benefits gained by fast reads disappear. In this way, the optimized fast-read cache can guarantee the lower bound for performance in the presence of frequent conflicts.

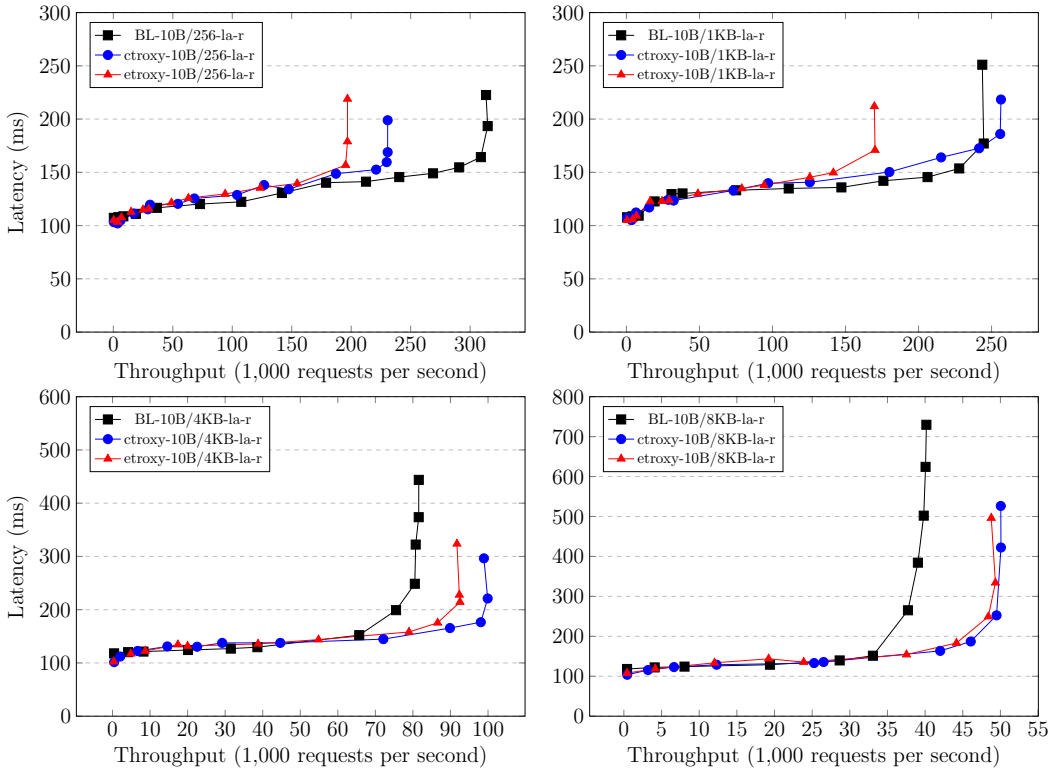



Figure 5.8: Read-only requests with a network delay of $100 \pm 20$ ms.

### 5.8.4 HTTP Service

In addition to the microbenchmark, we created a simple, replicated HTTP service that handles HTTP GET and POST requests and returns the queried or modified pages in response. The performance of the service is measured using the HTTP benchmarking tool Apache JMeter [130]. Since we are interested in evaluating the overhead of using a BFT system and a trusted subsystem in a latency-sensitive application, we ensure that JMeter is configured to not saturate replicas by launching 100 clients making a total of 500 requests per second.

We measure the performance of three implementations: (1) the baseline protocol; (2) an implementation of the Prophecy approach [24], a middlebox-based approach that mimics clients towards BFT replicas and is tailored to improve the performance of read-heavy workloads; and (3) TROXY. Table 5.1 summarizes these three implementations in terms of their read optimization approaches and consistency level.

The baseline protocol implements a PBFT-like read optimization that optimistically executes unordered read requests and accepts a result once $f + 1$ identical replies are received. In case of a failed quorum due to concurrent writes, the client must resend the request and ask for a regular ordering to enforce linearizability. Prophecy deploys a cache in a middlebox that sits between the client and the replicas. This cache stores the results of ordered reads to

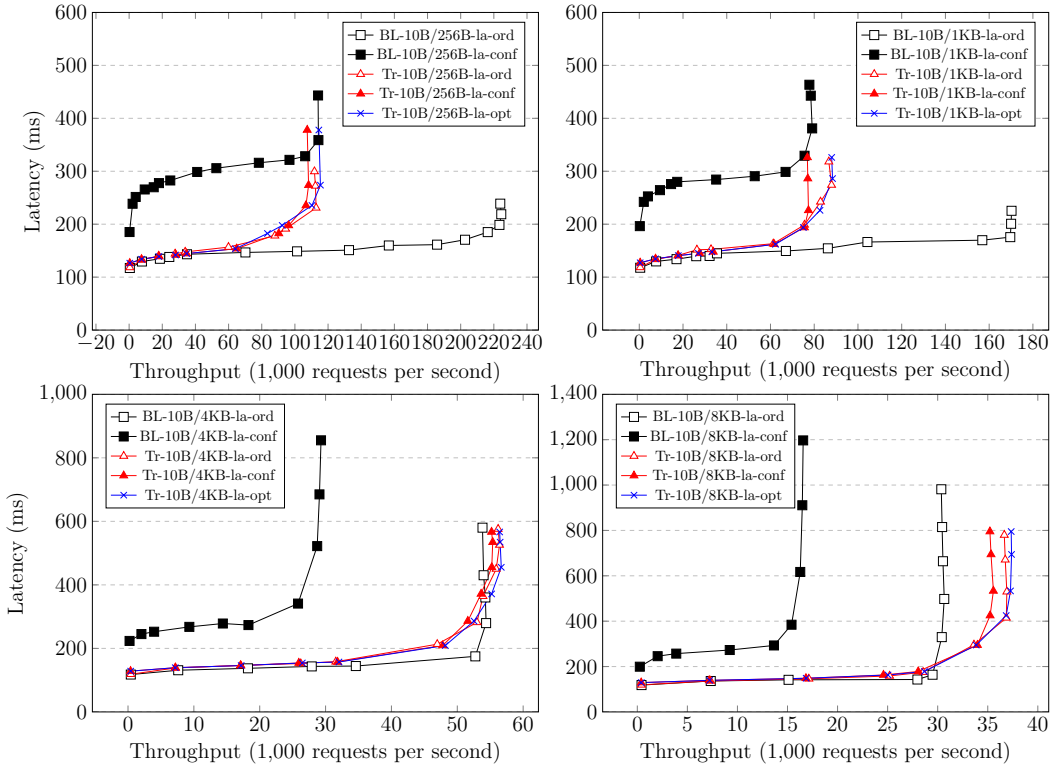



Figure 5.9: Read conflicts with a $100 \pm 20$ ms network delay.

Table 5.1: Summary of Read Optimization Approaches.

| | Replica | Quorum | Consistency |
|---|---|---|---|
| BL | $2f+1$ | $f+1$ replicas | Strong |
| Prophecy | $3f+1$ | 1 replica + middlebox | Weak |
| Troxy | $2f+1$ | $f+1$ replicas | Strong |

reduce the execution cost of read requests with large payloads for read-heavy applications. Only one reply if required from a randomly selected replica, which is compared to the cached result. However, consistency is traded off for higher throughput: the reply from a read operation reflects the state of the latest *read*, so in the worst case an outdated but correct result is returned to the client. In contrast, TROXY actively manages the fast-read cache to reflect the state changes caused by the latest *write*, and can thus guarantee strong consistency.

For the baseline, we run JMeter on the same machine as the client-side library, and use a local socket connection for message forwarding. As with Prophecy, JMeter runs on a separate machine and establishes a secure socket connection to the client machine where the middlebox resides. Since TROXY provides transparent access to the clients, JMeter can connect directly to the replicas without any modifications. In addition, we also run a standalone version of the HTTP service with Jetty (v9.4) [131] to show its original performance.

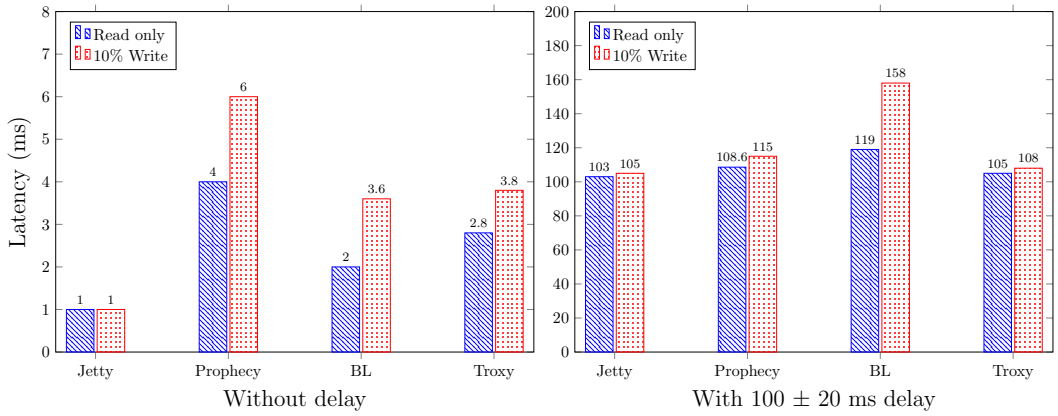



Figure 5.10: HTTP service in the local network and with a network delay.

The measurements are conducted in two scenarios: in the local network and with a network delay of $100 \pm 20$ ms. The requests GET and POST are issued with a payload size of 200 B, while the size of the reply messages ranges between 4 KB and 18 KB. The average request execution latency is shown in Figure 5.10. In both scenarios, the standalone implementation (Jetty) shows the original performance of the service. In the case of a local network, both the baseline and TROXY maintain low latency, with an overhead of at most 1.8 ms, while the two socket connections in Prophecy contribute to nearly twice the latency. When network delay is applied, the latency of the baseline implementation increases dramatically, as its reply voter, located on the client machine, must wait for $f + 1$ replies, so the observed latency is significantly affected by the slowest reply the reply voter receives. In Prophecy and TROXY, according to their design, the reply voter is not located at the client, but near the replicas (on the middlebox or in the fast-read cache of a replica), so the impact of this additional roundtrip between the reply voter and the replicas is negligible. The results of this measurement show that in a wide-area network, the use of TROXY-backed BFT systems is particularly beneficial for user-facing services.

## 5.9 Chapter Summary

In this chapter, we introduced TROXY, a system that offers transparent client access to BFT systems by leveraging trusted execution environments. Unlike traditional BFT systems, a TROXY-backed BFT system by design does not require any changes to clients to contain and execute a portion of the BFT library that contains essential functions such as reply voting and connection management. Instead, it creates a replacement of the library based on a trusted subsystem and allocates this trusted proxy within each replica. In addition, it introduces a novel read optimization that includes a managed fast-read cache to speed up read-heavy operations while providing strong consistency guarantees. According to the evaluation results, the use of TROXY can bring significant benefits to services with read-heavy workloads and a

relatively large reply size, especially in a wide-area network. This makes TROXY-backed BFT systems suitable for common Internet-based applications such as Web and e-mail services, where client modification is not an option due to the standardized protocols used and the variety of client implementations. Consequently, using TROXY helps solve the problem of traditional BFT systems not being suitable for user-facing legacy applications by removing the client-side library.

# 6

# Automated Deployment and Evaluation in the Cloud

Byzantine fault-tolerant (BFT) systems have made remarkable progress in terms of performance, scalability, accessibility, and so on. However, with all these improvements, the complexity of implementing such systems has also increased. During research on our previous works SAREK and TROXY, we found that the deployment and evaluation processes of an advanced BFT system can be quite time-consuming and error-prone. This leads to significant extra development effort, especially when there are special software and hardware requirements. In addition, it is difficult to make thorough comparisons between different BFT systems because, although they all run on a server cluster, the infrastructure of these clusters can be very different, and the impact of these differences remains unclear.

Therefore, in this chapter, we present the design for building a cloud environment-based framework that enables automated deployment and evaluation of different BFT systems. Since most existing clouds do not provide support for managing and coordinating replicated stateful services, we provide this framework that enables users to automatically deploy and replicate their applications with fault tolerance. Moreover, the proposed framework also automates the process of evaluating BFT systems by performing continuous testing with different system settings and configurations without manual intervention. We compared the Platform-as-a-Service (PaaS) cloud and the Metal-as-a-Service (MaaS) cloud and found that the MaaS cloud model is more suitable for building the infrastructure of the proposed cloud environment. While virtualization and container systems aim to increase the utilization of a single physical machine via multi-tenancy as in the PaaS cloud, a MaaS cloud takes a different approach by providing complete physical machines to users. This is particularly useful in our case as it facilitates the fulfillment of requirements, such as specific hardware or features, and also ensures that the resources of the machines are fully utilized by the deployed applications without interfering with other users. The prototype was evaluated with a benchmark that measures the elapsed time during the deployment process. The results are shown and discussed at the end of this chapter.

## 6.1 Relevant Cloud Models

In Chapter 2.2 we briefly introduced the background of the different types of cloud models. In this section, we select the Platform-as-a-Service (PaaS) cloud and the Metal-as-a-Service (MaaS) cloud based on their potential for building the infrastructure for the proposed cloud environment and explain their backgrounds, such as features and uses with examples. A comparison of these two models at the end of the section helps us to decide which model is more suitable for implementing a cloud environment that automates the deployment and evaluation processes of various Byzantine fault tolerant (BFT) systems.

### 6.1.1 Platform-as-a-Service Cloud

The provider of a Platform-as-a-Service (PaaS) cloud builds a resilient and optimized environment, and offers it to users to install applications and data sets. Within a PaaS cloud, infrastructure components such as operating systems, servers, databases, middleware, networking equipment and storage are incorporated by the provider, as are resources such as database management, programming languages, libraries and various development tools. All of these components are owned and maintained by the provider, allowing cloud users to focus on building and running applications rather than constructing and maintaining the underlying infrastructure and services.

Software development today can benefit significantly from PaaS cloud services. In addition to the infrastructure and services, many PaaS cloud platforms also provide useful tools such as version control, compilation and testing services that help developers to build and test new software faster and more efficiently.

#### 6.1.1.1 Operating System Virtualization

The implementation of a PaaS cloud is usually based on operating system (OS) virtualization, which is a lightweight server virtualization technology as opposed to full system virtualization. With OS virtualization, multiple isolated groups of user-space instances can coexist, allowing different users to simultaneously run different applications on a single machine. This ensures that the virtualized operating systems, although on the same machine, behave as individual systems without interfering with each other. As a result, they can accept commands from different users running different applications on that machine. It also monitors the interactions between the processes running within the virtualized environment and the underlying operating systems.

The virtualized user space instances are referred to as *containers*. To a program running inside a container, the container looks just like a real computing machine. In this case, the kernel of the host machine is shared by all containers. However, the contents of a container as well as the resources associated with the container (e.g., file system, connected devices) are visible only to the program running inside of it. Therefore, OS virtualization offers solutions for resource sharing, e.g., by implementing fair CPU time scheduling, file system isolation.

A state-of-the-art example of OS virtualization is Docker [132, 133] container technology, which is an open platform for rapid application development, deployment, and execution.

For example, a typical use case of Docker is to run a container for a web application or the supporting components such as a database server used by the web application. When the application running in a container needs to be updated, the existing container is removed and a new container is recreated based on the changed configuration of that application.

Docker containers are isolated from each other as well as from the host machine to ensure security. The degree of isolation with respect to network, storage or other underlying subsystems can be controlled with the deployment configuration. When deploying a system that contains multiple components or requires continuous deployment, the need for automation becomes critical, such as using a tool to manage provisioning, deployment, monitoring and resource balancing.

Container orchestration tools such as Docker Swarm [134] and Kubernetes [135] provide solutions for managing complex deployments. Both tools allow users to specify their requirements for a system and turn them into reality by managing container lifecycles and monitoring the health of containers and services. Although they share many similarities in their approaches such as using containers as units and offering crash-stop fault tolerance, the main differences are in the complexity and completeness of the approaches. Swarm mode makes it easy to set up and run containers in no time, using the same Docker command line interface as for building images or running containers. This brings great convenience to Docker users, allowing them to achieve productivity in a short time but with limited functionality. In comparison, Kubernetes has been developed for a longer time with more features, such as more commands and configuration settings, and is therefore used in many more use cases and conditions. With the completeness of the feature set and flexibility to handle arbitrary use cases, Kubernetes is more applicable but also more complex for the users to learn. Therefore, the major cloud service providers offer an off-the-shelf Kubernetes service in the cloud deployment scenarios, which significantly reduces the complexity of setup by the users themselves.

#### 6.1.1.2 PaaS Cloud Example: OpenShift

To explain the details of using a PaaS cloud to deploy applications, we take OpenShift [76] as an example, which successfully leverages Docker [132] container and Kubernetes [135] technology to achieve OS virtualization and container orchestration. OpenShift has been developed by Red Hat [136] since 2011 and currently has three main distributions:

- OpenShift Online [137]: a public cloud platform that provides on-demand access to compute resources for deploying and hosting cloud applications.
- OpenShift Container Platform (formerly known as OpenShift Enterprise) [138]: an enterprise version of OpenShift that gives companies full control over their cloud environments. It manages cloud-native and proprietary applications on a single platform and helps enterprises build and run applications anywhere.
- OpenShift OKD (formerly known as OpenShift Origin): a community distribution of OpenShift that provides a complete open-source platform for hosting applications in a container cluster and managing the application lifecycle.

In this thesis, we refer to the community distribution OpenShift OKD in the following text as OpenShift. The architecture of OpenShift can be represented as a multi-tiered system designed to expose the functionality of underlying components such as Docker [132] and Kubernetes [135] to an application management abstraction, as shown in Figure 6.1.

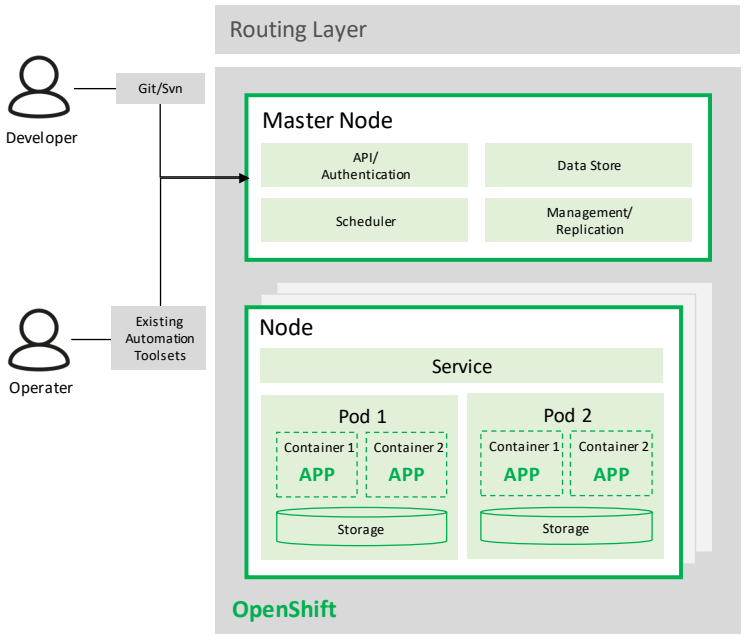



Figure 6.1: Architecture of OpenShift OKD.

As we can see from OpenShift's architecture, Docker provides the service for deploying one or more *containers* based on Linux-based lightweight container images. The containers run inside a Kubernetes *pod*, which is OpenShift's smallest compute and IP address-reachable unit. A pod owns its entire port space, and its storage and network are shared by all containers within it. When a pod is assigned to run on a host, its lifecycle begins and ends when it is eliminated or all of its containers exit. When containers run within a pod, they cannot be modified, so they are mostly immutable. Therefore, to alter the contents of an existing pod, it should first be terminated and then recreated with a changed configuration or container images. However, pods should be treated as transient and expendable, meaning they can be terminated at any time and do not retain state when recreated. This allows a higher-level controller to manage the pods, rather than directly by the users.

The concept of a *service* in Kubernetes is used for controlling the pods. A service usually consists of one or more replicated pods and acts as an internal load balancer. Typically, it runs like a proxy to identify pods by their labels with a label selector that finds all running containers that provide the desired service, so that the service can forward the received

connections and data to the correct containers. A service can be accessed internally by other pods using the service cluster's default IP address from OpenShift's internal network, and it also provides users with external access using external IP addresses assigned to it. Even if pods are arbitrarily added or removed from a service, the consistent IP addresses of the service guarantee that the service is consistently available and can always be reached by users. This allows us to create applications that are highly available and easily maintain load balancing between replicated pods.

In Kubernetes, a node provides the runtime environment for containers. Nodes in the Kubernetes cluster have the necessary services to run pods, including the container runtime environment, a kubelet, and a service proxy, as well as the services to be managed by the master node. The master node uses the information from node objects to validate nodes with health checks.

Users can inject secrets into a container via the service to which the container is attached. As for application developers, they can use the client command line tool (CLI) or the web console to communicate via RESTful API calls. OpenShift security is guaranteed by authenticating users with their credentials and authorizing them by determining whether they are allowed to perform a particular action based on their roles. Client operations are defined by the OpenShift policy engine and grouped into roles to handle user authorization. The policy engine checks the roles assigned to the user before allowing them to perform the operations. Additionally, Transport Layer Security (TLS) is used to secure communications between users and OpenShift as well as internal traffic, by supporting message encryption, data integrity and server authentication.

### 6.1.2 Metal-as-a-Service Cloud

Through virtualization and containerization, the multi-tenancy feature allows PaaS cloud services to increase the utilization of a single physical machine. However, this feature can cause problems in certain use cases for the following reasons: (1) A middleware layer of virtualization and orchestration, e.g., using Docker engine and Kubernetes, can significantly increase the risk of software errors or attacks. (2) In the event that the services provided require specific hardware or the resources of the entire host to run, such requirements can hardly be met by using virtual machines or containers.

Therefore, instead of sharing infrastructure resources, cloud providers take a different approach with Metal-as-a-Service (MaaS): when a customer requires compute resources, a MaaS cloud provider addresses the requirement by provisioning and delivering individual physical machines directly. The most modern MaaS cloud implementation is Ubuntu MAAS [139]. In addition to the physical machines delivered, customers can choose from various configuration management tools such as Ansible [140], Chef [141] and Puppet [142], to further automate the software deployment process.

#### 6.1.2.1 MaaS Cloud Example: Ubuntu MAAS

We take Ubuntu MAAS [139] as an example to explain how to build the cloud and deliver physical machines to users. Ubuntu MAAS is an open-source cloud service tool developed

by Red Hat for automated server provisioning and simple network setup on physical server machines. It is designed for flexible and efficient management of bare metal servers on a large scale, such as within a data center. The Ubuntu MAAS cloud model has a multi-tier architecture, as shown in Figure 6.2.

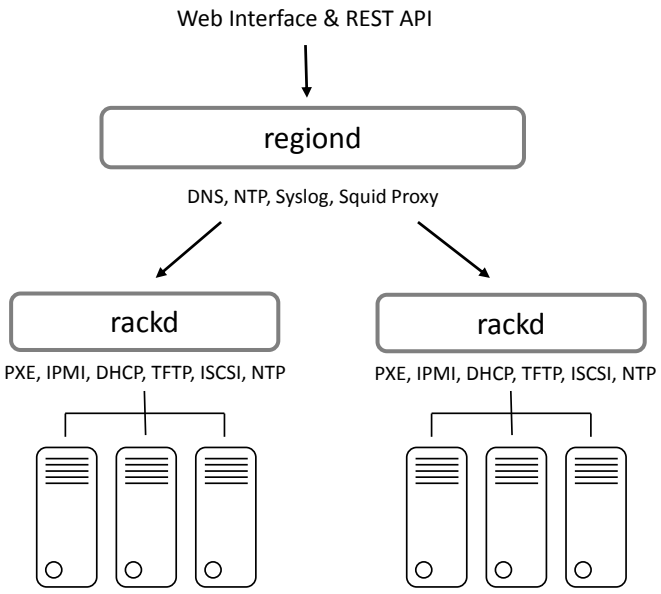



Figure 6.2: Tiered architecture of Ubuntu MAAS.

Ubuntu MAAS uses a central component called *Region Controller* (regiond) to handle requests, which is backed by a database, and provides a web interface and REST API for services such as DNS and NTP, to the Rack Controllers. This region controller has horizontal scalability and is stateless, that is, it does not manage any state except the credentials for communicating with the controller. Distributed *Rack Controllers* (rackd) provide DHCP (Dynamic Host Configuration Protocol), IPMI (Intelligent Platform Management Interface), PXE (Preboot Execution Environment), TFTP (Trivial File Transfer Protocol), and other services to multiple racks.

Ubuntu MAAS (referred to as MAAS in the following text) can group machine sets by rack or room and store large items such as operating systems at the rack level; it can also build a physical availability zone based on common points of failure. In MAAS, each machine is referred to as a *node* and is managed by MAAS through the following lifecycle:

1. Machines are configured for network booting so that MAAS can detect them. Once new machines connected to the MAAS network are detected, they are automatically added to the network.

2. When selecting machines, compute resources such as CPU, RAM, disks and NICs are listed and used as a constraint.

3. A successfully commissioned machine then becomes "Ready" and is configured for power control, e.g., to start/stop the machine or to apply/reassign a fresh operating system to the machine.

4. Machines that are ready can be offered to users and the network can also be configured.

5. Through MAAS, users can power on the assigned machines, install the desired operating system, configure the network, etc.

6. When a machine is no longer needed, it can be released back to the group of machines and later reassigned to other users. The previously installed operating system and client software are completely deleted from the hard disk during reassignment.

#### 6.1.2.2 Configuration Tool Example: Ansible

Compared to other widely used open-source configuration management tools such as Chef [141] and Puppet [142], Ansible [140] is much easier to set up because it only has a master running on the server machine, but no agents running on the client machines. Also, it is easier to learn to manage configurations in Ansible because it uses YAML (Yet Another Markup Language), a human-readable data-serialization language which resembles English, while others rely on their domain specific languages. Therefore, we take Ansible as an example to explain how to use an automation tool for software application provisioning, deployment, orchestration and configuration management.

A typical setup for using Ansible consists of two types of machines: a controller machine and worker nodes. Users start the orchestration on the controller machine to manage the worker nodes via a secure shell (SSH). We briefly explain Ansible's architecture by introducing its components as shown in Figure 6.3 and take an example of using Ansible to deploy a multi-tier application.

**Modules** Ansible describes how the components of an application interact in YAML scripts. These interrelated components are called "modules", each of which is a small program that contains the required state of the system. Ansible executes each module and then removes it.

**Plugins** Ansible uses plugins to extend its core functionality.

**Inventory** An inventory file represents the machines (e.g., AWS EC2 [143] virtual machines, OpenStack [82], or Ubuntu MAAS physical machines) that are added to groups and managed by Ansible, and contains the variables assigned to those machines.

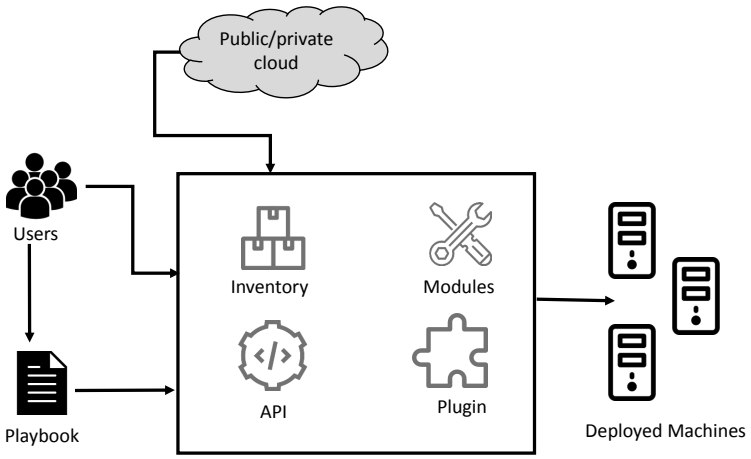



Figure 6.3: Architecture of Ansible.

**Playbooks** Ansible playbooks are used to manage the configuration of an application and orchestrate deployment to multiple machines. They can declare configurations, manually order processes, or even orchestrate interactions between different machines in specific orders during software deployment, in both synchronous and asynchronous ways.

**API** The Python APIs provided by Ansible are used to extend its connection types (in addition to SSH), callbacks (such as how to log) or new server behavior.

### 6.1.3 Comparison of PaaS and MaaS

We summarize the properties of PaaS and MaaS cloud models by making a comparison between them as shown in Table 6.1. To ensure that the proposed framework is able to solve the problem described in Section 3.1.4, we need to find out whether our cloud infrastructure requirements can be met. For example, to allow the use of different BFT systems, supports for the use of specific software or hardware are required; the installation of an evaluation setup should be relatively fast and efficient; the cloud infrastructure should not impose any constraints on the performance of the evaluated BFT systems, and so on. By showing the properties of both cloud models and relating them to our requirements, this comparison will help us gain insight into these two models and eventually lead to a decision on which model is more suitable for building the proposed framework.

A PaaS cloud provides users with a platform that includes both hardware and software tools that they can build upon and use to create custom applications. When using the PaaS platform, users are offered different levels of resources that can be easily scaled up or down based on virtualization technology as the business changes. All servers, storage, networking, etc. are managed by the cloud providers, and users have no control over the underly-

Table 6.1: Comparison of PaaS and MaaS.

| | PaaS | MaaS |
|---|---|---|
| Offered Services | Hardware and software tools | Hardware tools |
| Multi-tenancy | Support multiple users on the same platform | Support multiple users on different machines |
| Scalability | Various tiers of resources to suit the size of business | Application-driven resource scaling |
| Virtualization | Using virtualization | Without virtualization |
| User Control | No control of virtual machines | Full control of physical machines |
| Operational Capability | Operational limitation applied to end users | Support customized cloud operations |
| Tolerated service fault (stateless vs. stateful) | Stateless service | Stateless and stateful services |

ing virtual machines that process the data, only the management of their own applications. However, if the applications require specific runtime environments or hardware components, such requirements may not be met by the PaaS cloud because users are not able to develop their own dependencies with the platform. For example, the deployment of BFT systems that assume a hybrid fault model (see Section 2.1.2.2) usually requires a trusted execution environment (TEE), which is hardly available in existing PaaS cloud solutions. Moreover, fault tolerance of stateless services is well supported by PaaS clouds with horizontal scalability, but cannot be applied to stateful services. In contrast, a MaaS cloud mostly just provides hardware tools in the form of physical machines, so users can rent as many dedicated servers as they want and no longer have to deal with the abstraction layer provided by virtualization. This means that users gain full control over the low-level hardware architecture, custom software applications, and everything in between. Therefore, the specific runtime/hardware problem with PaaS solutions can be easily solved when MaaS solutions are used to build the cloud infrastructure. This also enables fault tolerance of stateful services for MaaS solutions as they are now able to integrate Byzantine fault-tolerant systems to achieve this.

In a performance evaluation, we typically expect the evaluated system to reach its peak performance when the compute resources on the replica machines are saturated. Since both PaaS and MaaS solutions allow multi-tenancy, i.e., the resource pool is shared among all users, it is necessary to find out whether this feature would limit the performance of the evaluated systems. As with the PaaS solution, the performance of an application running in a virtual machine may be affected by other applications as they may share the same underlying hardware or network resources. In this case, using the MaaS solution brings another benefit: no more sharing resources with other users, as the physical machines provided are dedicated to the users. This provides a significant speed increase compared to sharing resources at the virtual machine level and, more importantly, prevents interference from other users when performing evaluations.

The proposed framework should be able to quickly install evaluation setups and efficiently create replications for fault tolerance, which assumes that the underlying cloud infrastructure supports customized cloud operations with management automation. However, this may not be true for PaaS solutions, as the PaaS platform tends to limit the operational capabilities of end users. Although this should reduce the operational overhead for users, the loss of operational control makes it difficult to integrate and use automation tools. On the other hand, custom cloud operations can be fully supported by MaaS solutions by leveraging automation tools for the deployment as well as the evaluation processes.

Considering the above properties of both cloud models and the actual requirements of the proposed framework, as a result of this comparison, the MaaS cloud solution is considered more suitable for building an infrastructure for the proposed framework.

## 6.2 Related Works

In this section, we give an overview of previous works aimed at providing an environment where BFT protocols are integrated, configured, and can be compared under certain conditions. In this context, we also compare our work with some of the existing frameworks.

### 6.2.1 BFT Systems in the Cloud

Many research works [144, 145, 146, 147] have been proposed to integrate BFT protocols into cloud infrastructures to tolerate Byzantine failures. For example, Bessani et al. have presented a virtual storage cloud system called DepSky [144], which consists of a combination of different untrusted cloud storage services to build a cloud-of-clouds. It improves the availability and confidentiality provided by cloud services by combining Byzantine fault tolerance with other techniques such as erasure codes and cryptographic secret sharing. Similarly, Garraghan et al. [146] presented a BFT framework that enables the deployment of applications in a federated cloud environment to improve reliability by leveraging the inherent fault independence of the federated cloud. FITCH [145] presents a fault-tolerant cloud infrastructure to support dynamic adaptation of replicated services in cloud environments. It has been evaluated with two cloud services, which shows that it can reconfigure and adapt services to dynamic workloads, e.g., by adding/removing or upgrading/downgrading replicas. Also, at the middleware level, BFT has been combined with multi-tier web services to ensure their heterogeneous reliability requirements [116]. In contrast to them, our work focuses on enabling automated deployment of replicated applications through the use of configuration management tools.

### 6.2.2 Evaluation of BFT Systems

BFTsim [148] is a simulation framework that uses the network simulator [149] to investigate the effects of certain network conditions such as slow connections between a subset of all replicas. It implements the logic of the protocols being evaluated in a declarative networking language (OverLog), therefore the evaluation results reflect the essence of the protocols

and are not affected by implementation-specific factors. This requires that all factors affecting performance are identified and modeled in the simulation, but this is very difficult to achieve when the protocols use specific hardware components, such as CheapBFT [34] and HYBSTER [35]. Moreover, integrating a new protocol into BFTSim by implementing it in OverLog causes a significant porting effort, since most BFT prototypes are implemented in languages such as C/C++ or Java.

Similar to BFTSim, the Turret platform [150] also uses the network simulator [151] to emulate the network between clients and replicas. With Turret, the evaluation of a BFT protocol is done by running its prototype in virtual machines running on the same physical server. The platform is specifically designed to automatically identify vulnerabilities in BFT prototypes that could be exploited by malicious attackers for performance attacks. However, just like BFTSim, Turret cannot provide realistic performance results due to the simulated network between nodes. Consequently, neither of them is suitable for performing a comprehensive and comparative evaluation of different BFT systems. Unlike them, the proposed framework uses physical machines to set up the infrastructure and perform evaluations. This guarantees that the performance results are not affected by the virtualization layer, and allows meeting specific hardware requirements without having to rebuild or modify the BFT prototypes.

Instead of simulating the network, BFT-Bench [152] distributes the evaluated BFT systems across a cluster of physical machines. Technically, this approach should be able to generate performance results that resemble the behavior of a BFT system in a realistic application scenario. However, the results published so far indicate that BFT-Bench has significant limitations in generating the workload with its client emulator: It can only successfully run experiments at a relatively low throughput of less than 10,000 requests per second and can only support a few concurrent clients. In contrast, the proposed framework places no constraints on the evaluation setup and benchmarks, and can furthermore automate the deployment and evaluation processes for the BFT prototypes.

### 6.2.3 Automated Deployment of BFT Systems

Lazarus [153] was introduced as a control plane for automatically managing the deployment and execution of BFT systems with diverse replicas in terms of different operating systems and runtime environments. It continuously monitors the possible vulnerabilities of the replicas and measures the risk of the BFT system being compromised. It maximizes the failure independence by reconfiguring the set of replicas when the risk increases, handling reconfiguration process automatically. However, unlike our proposed framework, Lazarus' deploy manager is based on the Vagrant provisioning tool and VirtualBox, and relies on virtualization to create and deploy replicas.

In our previous work we proposed BFT-DEP [53], a framework for automatically deploying BFT protocols and applications in a PaaS cloud. As an extension of the PaaS cloud model, BFT-DEP leverages the functionality of the existing PaaS platform to address specific deployment requirements and provides tailored support for setting up and managing replicated services. By creating a BFT agreement layer with containerization, BFT-DEP integrates the

BFT protocol into the PaaS platform software stack as a built-in service, without requiring any changes to the cloud itself. In this way, the BFT protocol instances can be deployed and managed in the same way as other service instances in the PaaS platform, leveraging existing cloud facilities as much as possible and making them largely immune to changes in the cloud architecture. Once the BFT protocol instances are created and configured to set up the agreement layer, BFT-DEP offers a dedicated template for users to automatically create and deploy replicated applications. BFT-DEP guarantees that application replicas with their associated BFT protocol instances are distributed across different hosts of the cloud infrastructure to enforce fault tolerance, and handles coordination among replicas. An initial realization of BFT-DEP was implemented as an extension of OpenShift [76] based on a cluster of AWS EC2 instances, and integrates the BFT-SMaRt [154] protocol into the agreement layer. Therefore, BFT-DEP still has the problem that it relies on virtualization to build the cloud infrastructure and does not provide support for specific hardware usage of BFT systems, which can be solved by the proposed work we discuss below.

## 6.3 System Design and Implementation

### 6.3.1 Automated Deployment Framework

In section 6.1.2.1 and 6.1.2.2, we have briefly described how to use (1) Ubuntu MAAS for server provisioning, and (2) Ansible for application deployment and task management, such as automating server allocation, operating system installation, and application deployment. In this section, we present the system design for building a framework with these tools to automate deployment. Both Ubuntu MAAS and Ansible require a controller machine and use command line interface tools (CLI). In the following, we show a demonstration of the control flow of automated server deployment and the use of predefined shell scripts to perform the deployment. Moreover, the solution for deploying different BFT prototypes is an important factor of the framework design. Therefore, we divide the whole process into four phases and discuss each of them separately.

#### 6.3.1.1 Server Provisioning Phase

Figure 6.4 shows the setup and workflow of the server provisioning phase, where a controller runs on the host machine of Ubuntu MAAS with multiple bare-metal machines. As mentioned in section 6.1.2.1, the controller machine can use the DHCP service provided by Ubuntu MAAS to set up the bare-metal machines that have enabled booting over the network as the first boot option, and include them in the managed cluster. All bare-metal machines that are part of the managed cluster are added to a resource pool during a commission, and then switch to the "Ready" state.

We created a *controller* consisting of a set of shell scripts, and run the scripts to manage Ubuntu MAAS operations on the host machine. At step ①, the controller queries the Ubuntu MAAS server to see if there are enough bare metal machines in the ready state. If the query is answered with a positive reply, the controller issues commands to Ubuntu MAAS to deploy

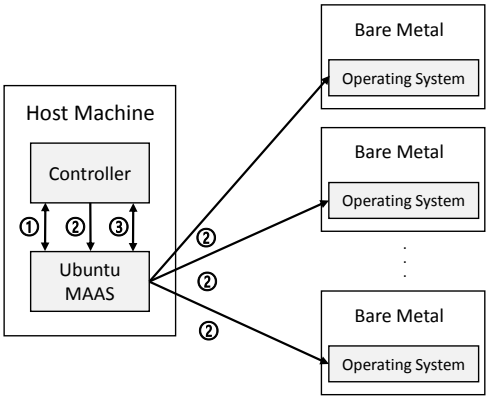



Figure 6.4: Provisioning servers from bare metal machines.

operating systems to the bare-metal machines in the ready state, as in step ②. During the deployment, the controller queries the Ubuntu MAAS server to check the status of the corresponding machines (③). Finally, this phase is completed after all machines have become "Deployed".

#### 6.3.1.2 Runtime Environment Deployment Phase

The second phase focuses on the deployment of the runtime environment, which follows the design shown in Figure 6.5.

At the end of the last phase, all bare-metal machines have operating systems installed and are "Deployed". Before we can start performance evaluation of the BFT prototypes, we also need to install the runtime environment required by the prototypes. Therefore, Ansible is needed to manage the configuration for installing the required runtime environment (e.g., Java runtime environment) and automate the deployment process. Since Ansible does not require agents on the worker nodes, it only runs on the same master machine that runs the Ubuntu MAAS server and controller, and connects to the worker nodes via SSH by default. To use Ansible, we first need to define the essential tasks and include them in the Ansible playbooks. In the next step, the controller sends a request to Ansible (①) to trigger the execution of the playbooks, which then install all the required components of the runtime environment for running and evaluating various BFT prototypes.

#### 6.3.1.3 BFT Protocol Deployment Phase

The main task of this phase is to deploy the prototypes of the different BFT protocols and their benchmarks, following the procedure shown in Figure 6.6.

The operations in this phase are also defined in the Ansible playbooks. In step ①, the controller script fetches the source code of BFT protocols from their remote repositories (e.g.,

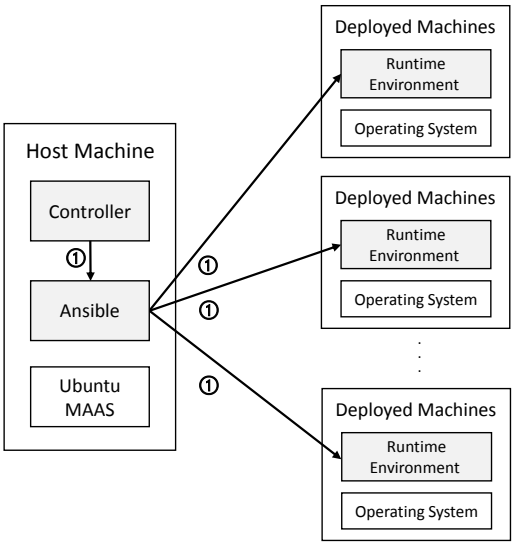



Figure 6.5: Deployment of the runtime environment.

using Git) and stores the code on the host machine. The next step (②) is to configure the code and save it for deployment. Downloading and configuring the code directly on the host machine instead of each replica can significantly reduce the time required for code distribution and configuration, since in the latter case such tasks can only be performed serially in the playbooks script. In the final step (③), the controller sends a command to Ansible notifying it to perform the deployment. Ansible starts executing the predefined playbooks to safely deliver the configured code to each deployed machine. It then compiles the code locally to install the BFT protocol, to ensure that if the code has specific compilation requirements, they can be met.

#### 6.3.1.4 BFT Protocol Evaluation Phase

In the final phase, the controller starts running the benchmarks against the deployed BFT protocols for evaluation, as shown in Figure 6.7.

There are two steps in this process: In step ①, the controller script asks Ansible to launch an instance of the deployed BFT protocol on each deployed machine as a server replica. Once all replicas are up and running, the controller proceeds to step ② and creates the client instances, for example, directly on the host machine. If a dedicated machine or more than one client machine is needed, they can be selected from the deployed machines and managed by Ansible in the same way as the server machines. The client instances then start running the benchmarks, and finally analyze the results.

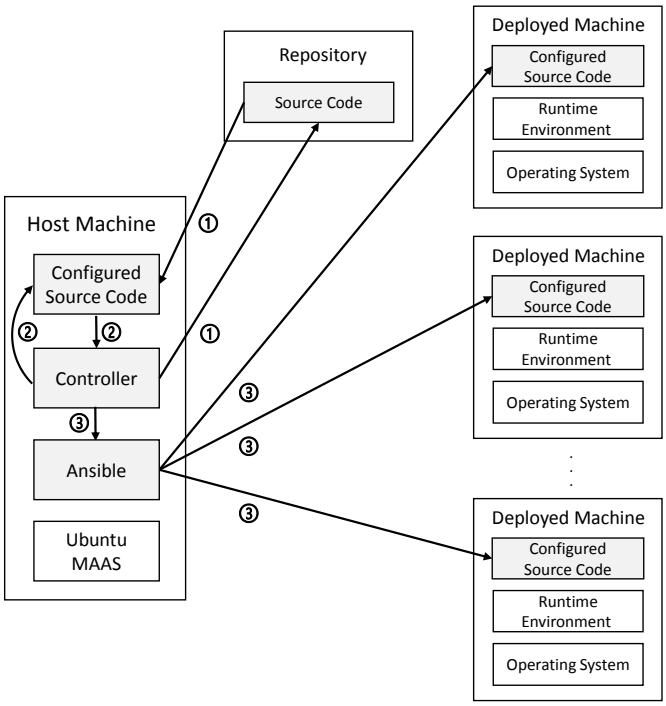



Figure 6.6: Deployment of BFT protocols and benchmarks.

### 6.3.2 Framework Implementation

We implemented the framework based on the design presented in the previous section. Figure 6.8 shows the structure of the main components of the controller on the host machine. In the following, we elaborate on the main components.

**Source Code** The directory for storing the source code of the various BFT prototypes. We selected three BFT prototypes for evaluation: BFT-SMaRt, TROXY, and HYBSTER. Since the implementation of TROXY is based on HYBSTER, both share the same directory and can be configured to run both systems. This directory can be extended to include more prototypes, by simply adding the source code of the new protocols to the subdirectory reserved for other prototypes.

**MAAS Deployment** The main script file and the entry point of the framework. We created the main script to drive the entire deployment process, including tasks such as deploying operating systems to the bare metal machines, using Ansible to deploy the runtime environment, and the BFT prototypes.

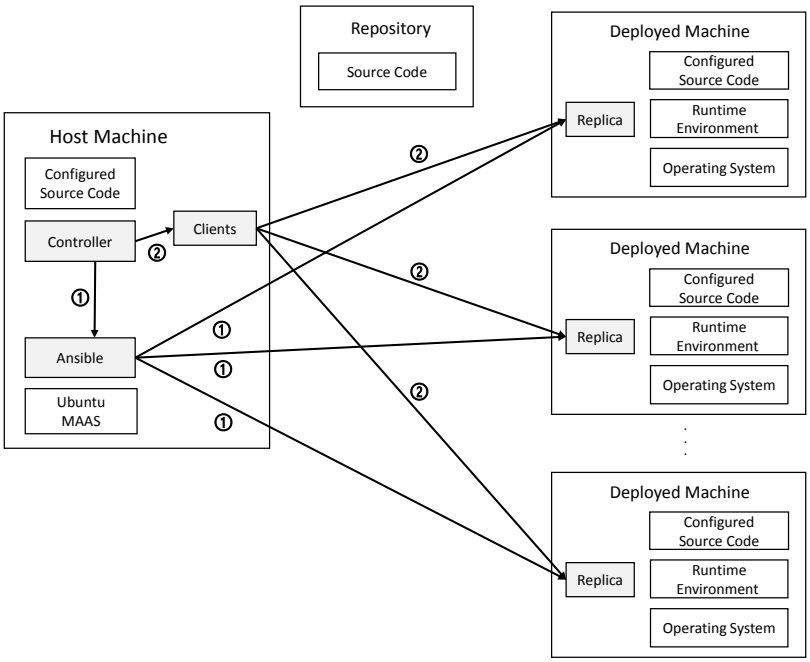



Figure 6.7: Evaluation of BFT protocols with benchmarks.

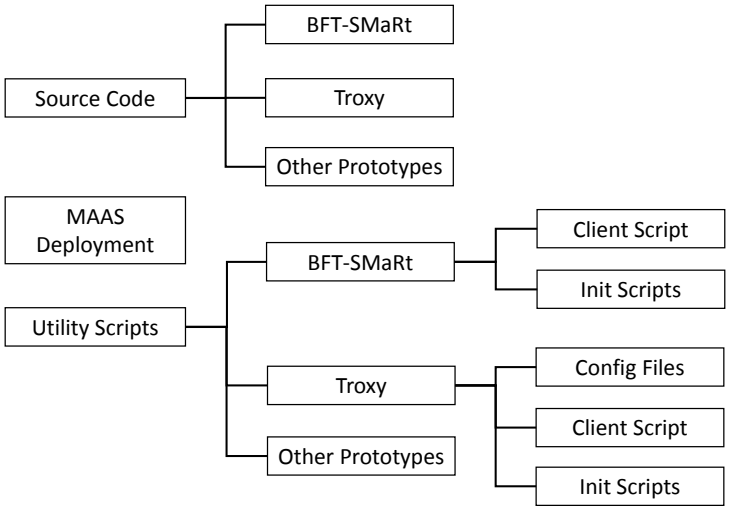



Figure 6.8: Structure of the controller.

**Utility Scripts** The directory for all scripts created and used to set up and perform evaluations. The structure of this directory is similar to the source code directory, with each BFT prototype having its own subdirectory. Within each subdirectory, two types of scripts are typically required: (1) *init scripts* to complete the setup of the prototypes, and (2) *client scripts* to run the replicas with benchmarking clients and finally to produce the analysis of the results. As mentioned earlier, since TROXY and HYBSTER share the same source code directory, additional configuration scripts are needed to set up and switch between these two prototypes. To evaluate a new BFT prototype, it is as simple as adding its utility scripts to a subdirectory created for that prototype.

## 6.4 Evaluation

We performed evaluations against the implemented framework using two benchmarks. The first is a time-cost benchmark that measures the time required for each step of the deployment process. In particular, we are interested in the time cost of deploying with different configurations on the same set of bare-metal machines, e.g., by installing different operating systems and BFT prototypes. By running this benchmark, we can gain a lot of insights into the time cost of each deployment phase and the impact of different configurations, which can be used to optimize the BFT prototypes. The second benchmark is a microbenchmark that measures the performance of two BFT prototypes on the machines that are automatically provisioned and deployed with the essential software stack. This benchmark allows us to determine (1) whether the measurements can be performed automatically by the proposed framework, and (2) whether the proposed framework is able to successfully run experiments, on the automatically deployed machines with a comparable performance result. Note that the result numbers and figures are automatically generated using predefined scripts managed by Ansible. Once the host machine has collected all the required data from the evaluations, it first runs the appropriate Ansible playbooks to start analyzing the collected data, and then uses the plotting scripts to visualize the data based on the analysis.

To conduct the evaluations, we used a cluster of five physical machines interconnected by a 1000 Mbps switch, with the following hardware configuration:

- One host machine: a 1000 Mbps Ethernet NIC, Intel Core i7-4790 quad-core CPU running at 3.60 GHz, 16 GB memory.
- Four replica machines: four 1000 Mbps Ethernet NICs, SGX-capable Intel Core i7-6700 quad-core CPU running at 3.40 GHz, 24 GB memory.

### 6.4.1 Time-Cost Benchmark Setup

We created this benchmark to measure the time it takes to execute each step of the deployment process. We particularly focus on the cases where the deployed machines use different software configurations. For example, the combination of different operating systems, runtime environments with different BFT prototypes can lead to different time costs for each

step of the process, so it is necessary to compare the results of different configurations to find out and analyze the differences caused by these factors. Analyzing the results of this benchmark can help BFT protocol developers gain insight into the impact of each factor on the overall time cost, which can be used to further improve the design and implementation of their prototypes and optimize the deployment process. It also helps us to find ways and means to reduce the time required to automate the deployment process and eventually improve the proposed framework.

The time-cost benchmark is performed as follows: (1) During the operating system and runtime deployment phase, the start and end time of each step where Ubuntu MAAS interacts with the bare metal machines are recorded. (2) During the deployment phase of the BFT prototype, a timer plugin of Ansible is used as soon as the execution of the Ansible deployment playbooks starts. Upon completion of the benchmark, all recorded data is transferred to the host machine where it is further analyzed to generate graphs for comparison.

The host machine has Ubuntu 16.04 with 64-bit and kernel 4.4.0 as the operating system, as well as the latest version of Ubuntu MAAS and Ansible installed. As part of the benchmark setup, we use different software configurations for the replica machines. We chose two major Linux distributions, 64-bit Ubuntu 16.04 (with a kernel 4.4.0) and 64-bit CentOS 7 (with a kernel 3.10.0), as operating systems, since both distributions are stable and widely used, but based on different architectures (Debian architecture for Ubuntu, Red Hat Enterprise Linux (RHEL) for CentOS). Three BFT prototypes are selected: BFT-SMaRt, TROXY, and HYBSTER. For the evaluation, we combine the operating system options and BFT prototypes in the following scenarios:

- **3 Ubuntu + 1 CentOS with BFT-SMaRt:** including four physical machines, three running Ubuntu and the other running CentOS. The BFT-SMaRt prototype is installed on all machines.
- **4 Ubuntu with BFT-SMaRt:** including four physical machines, all running Ubuntu and the BFT-SMaRt prototype.
- **2 Ubuntu + 1 CentOS with TROXY:** contains three physical machines, two running Ubuntu and the other running CentOS. All machines have the TROXY prototype installed.
- **3 Ubuntu with TROXY:** contains three physical machines, all running Ubuntu, with the TROXY prototype installed.
- **3 Ubuntu with HYBSTER:** contains three physical machines, all running Ubuntu, with the HYBSTER prototype installed.

In each scenario we measure the time required to complete the process of deploying operating systems and BFT prototypes. The entire deployment process can be divided into 8 steps:

1. **Obtain ready machines**: checks if there are enough bare metal machines in the ready state.

2. **Find IDs**: finds the IDs of the bare metal machines in the ready state.
3. **Deploying OS**: based on the IDs, start deploying operating systems to the machines.
4. **Deployment succeed**: queries the status of the machines until they all become "Deployed".
5. **MAAS Connection**: connects MAAS and Ansible by adding the IDs of the deployed machines to the Ansible configuration files.
6. **Modify CentOS configuration**: modifies the default CentOS SSH configuration to avoid errors when using Ansible.
7. **Execute init scripts**: creates tasks to download the source code and network environment configuration on the deployed machines and prepare the benchmarking clients.
8. **Execute playbooks**: invokes Ansible to run the predefined playbooks to complete the tasks for deploying the BFT prototypes.
9. **Total**: computes a summary of the time cost from the steps listed above.

We further break down the final step of running Ansible playbooks to deploy, compile, and run a BFT prototype into 5 steps:

1. **Pre-prepare runtime environment**: configures Python and other modules on the deployed machines to prepare to deploy the required runtime environment.
2. **Deploy runtime environment**: runs Ansible playbooks to deploy the specific runtime environment required by the BFT prototype.
3. **Download code**: transfers the source code of the corresponding BFT prototype to the deployed machines.
4. **Compile TROXY**: compiles TROXY and HYBSTER with SGX SDK.
5. **Start replicas**: starts running replicas on the deployed machines and clients on the host machine to complete the evaluation.

### 6.4.2 Time-Cost Benchmark Results

In this section we analyze the evaluation results collected by running the benchmark in different scenarios as previously defined (see 6.4.1), by summarizing the results into three comparison groups. Note that all evaluation data is collected automatically by running predefined Ansible playbooks after each run, and the final result is computed as the average of five runs. A plotting script is used to generate the graphs based on the final result and complete the evaluation. In all result plots, the *x* axis shows the time cost, measured in seconds, and is interrupted to ensure that large values can fit within the scale of the plot. The *y* axis shows each step of the deployment phases, as explained in Section 6.4.1.

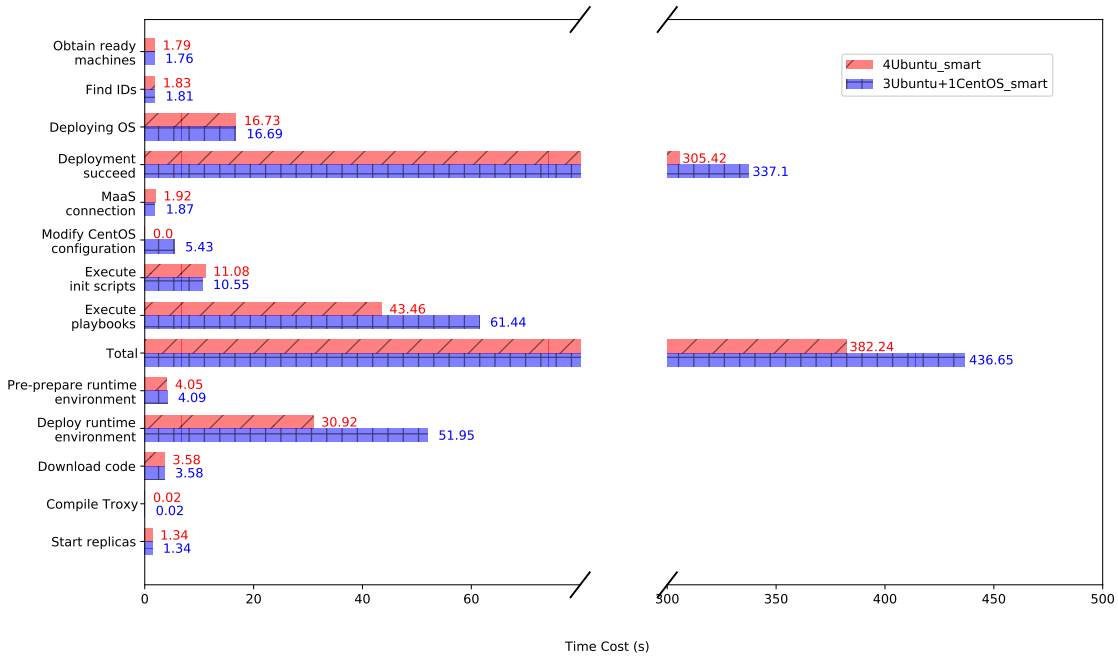



Figure 6.9: Time spent provisioning machines with different operating systems and deploying BFT-SMaRt.

For the first set of comparisons, we chose the time costs of the 4 Ubuntu with BFT-SMaRt and 3 Ubuntu + 1 CentOS with BFT-SMaRt scenarios, as shown in Figure 6.9. When provisioning the bare-metal machines, the first two steps require almost the same amount of time in both scenarios, as they only involve Ubuntu MAAS. The third step of deploying operating systems also does not result in a significant difference in time costs. Ubuntu MAAS takes little time to check several configuration files to acquire the bare-metal machines based on their IDs and confirm the specified distribution of operating systems on the acquired machines. Once Ubuntu MAAS starts installing the specified operating systems in step 4, the hybrid scenario takes about 30 more seconds to complete the installation, which is due to the fact that CentOS takes longer to install than Ubuntu, as we observed in the evaluation. To avoid connection failure when Ansible is later used to connect to the deployed machines, the default Generic Security Service Application Program Interface (GSSAPI) authentication option in CentOS must be changed on the appropriate machine, accounting for an additional 5 seconds in the hybrid case. While creating the tasks for deploying BFT prototypes takes a similar amount of time in both scenarios, executing these predefined playbooks leads to major differences. As we mentioned earlier, the 5 steps after "Total" serve as a breakdown of the "Execute playbooks" step. They help us to find out the reason for the differences in the time taken, which is mainly related to the deployment of the runtime environment on the machines running different operating systems. Normally, when using Ansible's playbooks-based automation, all tasks are added to the playbooks and later executed serially. In a

homogeneous cluster like in the 4 Ubuntu with BFT-SMaRt scenario, the processes to deploy the required runtime environment on the machines with the same operating system can actually be executed in parallel. However, once the runtime deployment process for one operating system is started, the same process for another operating system has to wait until the previous one completes. This is caused by Ansible's limitation that these two tasks must be scheduled serially in the playbooks. As a result, in the hybrid scenario, it takes longer for the deployment process to complete. From this comparison, we can conclude that in evaluations, a homogeneous system might be more efficient than a hybrid system in terms of time cost.

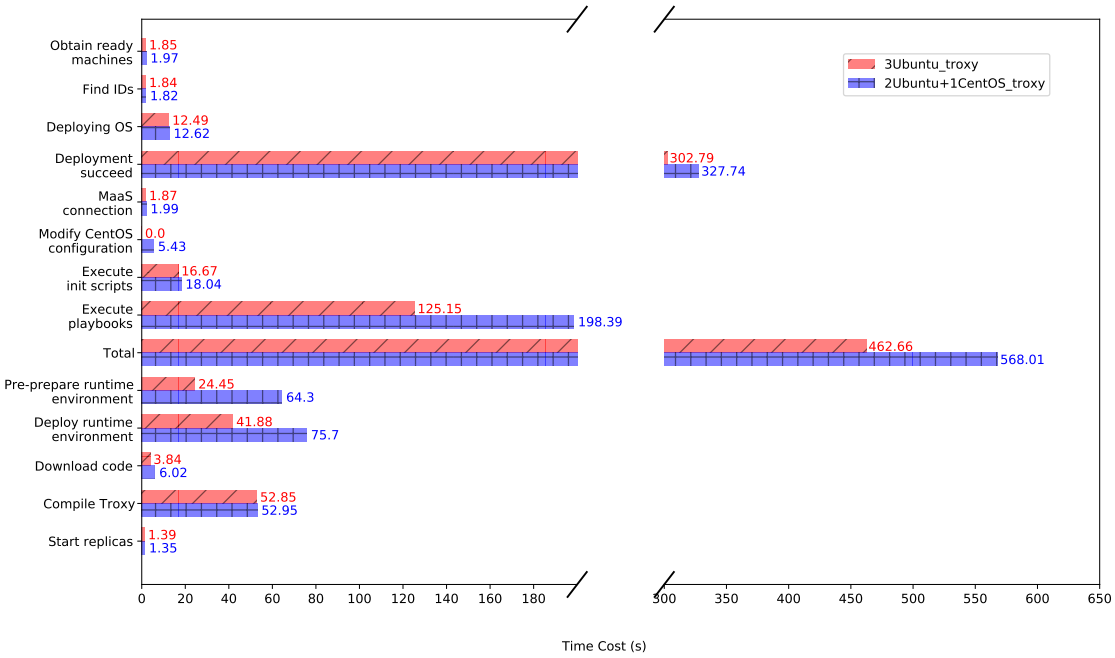



Figure 6.10: Time spent provisioning machines with different operating systems, deploying and compiling TROXY.

For the second set of comparisons, we analyze the difference in time cost between the scenarios 3 Ubuntu with TROXY and 2 Ubuntu + 1 CentOS with TROXY. The result shown in Figure 6.10 indicates that deploying the BFT prototype on a homogeneous system takes much less time, especially when specific runtime environments are required, e.g., the SGX SDK and other external libraries to compile and run TROXY (see Section 5.7). As for the difference in time cost for each step, in addition to those already discussed in the previous comparison, we found that in this case it is also quite large for the preparation phase of the runtime deployment. The main reason for this is that the more requirements a BFT prototype has for the runtime environment, the more preparation it needs when configuring the base modules. As a result, both lead to an increase in serialization of execution tasks when using Ansible for a hybrid system.

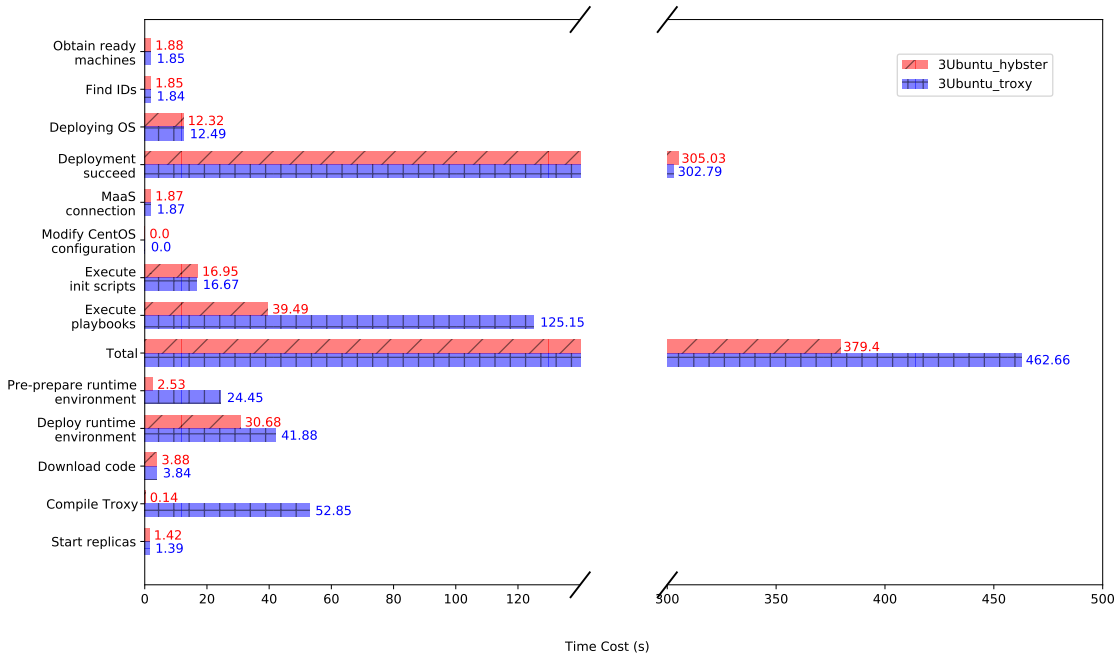



Figure 6.11: Time spent provisioning machines with the same operating system, deploying and compiling TROXY and HYBSTER.

The last comparison group consists of two homogeneous systems, but on which different BFT prototypes are deployed. Thus, we can compare the time cost of deploying a cluster with all Ubuntu machines with HYBSTER and TROXY, as shown in Figure 6.11. With the same operating system, the steps that are independent of the BFT prototypes have almost the same time cost in both scenarios. A big difference can be observed when it comes to the steps where Ansible playbooks are executed for runtime environment installation and code compilation. Since TROXY requires more external libraries to be installed and configured, it takes much longer to prepare, deploy and compile the required runtime environment and libraries.

### 6.4.3 Microbenchmark Setup

We also created a microbenchmark to evaluate the performance of the two BFT prototypes TROXY and HYBSTER on the machines automatically deployed by the proposed framework. All the evaluation processes, as well as the analysis of the result data, are defined as tasks in the playbooks managed and operated by Ansible, so that they can be executed after the machine deployment is completed. Regarding the evaluation setup, we configure the proposed framework to ensure that the automated deployment process results in an identical setup as the one performed for the microbenchmark in Section 5.8.3. On the client side, a configured number of clients are created to continuously issue asynchronous requests to the replicas over secure socket connections. On the server side, all replicas are configured

to run on Ubuntu 16.04 with 64-bit kernel 4.4.0, OpenJDK 1.8 and the Intel SGX SDK v1.9. Other configurations of the setup such as secure connections and message authentication are kept the same as in Section 5.8.3. We create tasks for Ansible to measure the average throughput and latency for 60 seconds, and the final results are formed from the averages of five runs. By running this benchmark, we can verify the ability of the proposed framework to successfully run experiments and perform evaluations with comparable results.

### 6.4.4 Microbenchmark Results

In this benchmark we created the request and reply messages with the same payload size in three settings: 256 B/256 B, 1024 B/1024 B and 4096 B/4096 B. Optimization approaches such as fast-read or batching are not used since they are orthogonal approaches. The performance of the HYBSTER prototype is compared to the *etroxy* implementation of the TROXY prototype running inside an enclave and adding the overhead of using the trusted subsystem. All results are measured in the local network environment with no additional latency. Once all evaluations are complete, Ansible runs the predefined scripts to collect and analyze the result data, and finally generates the plot shown in Figure 6.12.

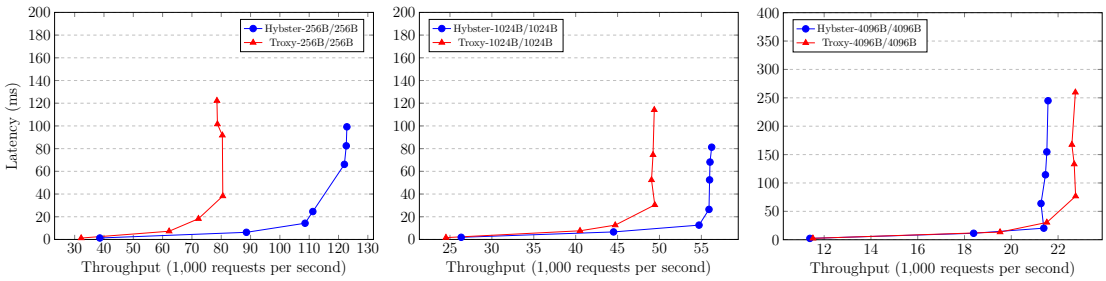



Figure 6.12: Automated evaluation of write requests with TROXY and HYBSTER.

First of all, Figure 6.12 shows that the proposed framework is able to successfully perform automated evaluations on the deployed machines. It has managed to continuously measure the performance under different settings and handle the pipeline of data collection and analysis. Associated with a given workload, the evaluation results also show that they are comparable to the results of manual tests. For example, for a small payload size of 256 B for request and reply messages, TROXY shows a performance loss of about 50% due to the overhead of transitions between trusted and untrusted environments, which is consistent with our observation in Section 5.8.3. As the payload size increases, the performance of TROXY improves significantly and reaches the same throughput as HYBSTER at 4096 B, which is also indicative of the same trend we saw earlier.

## 6.5 Chapter Summary

The deployment and evaluation of BFT protocols with replicated services have not been well supported by existing cloud environments due to many usability and performance limita-

tions. In this chapter, we presented and evaluated a framework for automated deployment and evaluation of BFT prototypes based on a Metal-as-a-Service (MaaS) cloud. With the MaaS cloud deploying physical machines, we use an orchestration tool Ansible to automate the process of installing and configuring the essential software stack and BFT prototypes. The evaluation process and data analysis are performed automatically by executing scripts with predefined tasks, without requiring any commands during runtime. The time-cost benchmark accurately measures the time required for each step of the deployment process, which clearly demonstrates the benefits of automated management in terms of saving time and avoiding operational errors. It also provides insights into the impact of different configurations on the deployed software stack and BFT prototypes to help users optimize their systems. We also conducted a simple microbenchmark to validate the applicability of the proposed framework, that it can successfully perform automated evaluations on the deployed machines and produce comparable performance results. For future work, more BFT prototypes could be added to the framework and the benchmarks can be extended with more settings.

# 7

# Conclusions and Further Ideas

In this thesis, we have investigated how to improve the performance, reliability and usability of Byzantine fault-tolerant (BFT) systems, and presented a solution for automated deployment and evaluation of BFT systems in a cloud environment. In this chapter, we summarize the content presented in the previous chapters and their contributions. We also outline the directions for further research based on this thesis.

## 7.1 Conclusions

Byzantine fault-tolerant (BFT) systems have been proposed to guarantee the reliability of replicated stateful services across replicas. With the rapid development of cloud computing in recent years, more and more services with high fault tolerance requirements are gradually migrated to the cloud environment, e.g., transferring data and applications from an on-premises data center to the public cloud. As a result, BFT systems are considered essential for these services, as they can tolerate not only crashes, but also software errors and even malicious attacks. So far, however, we do not see BFT systems being widely adopted by industry for use in production.

**Research Questions** In this thesis, we have identified three main problems with existing BFT systems:

- **Limited performance** has been identified as one of the main problems of the existing BFT systems, which is mainly caused by using only a single leader to maintain the total order of all requests, introducing additional overhead into the leader replica and thus affecting the system performance.
- The **client-side BFT library**, which contains essential functions, has nevertheless become an obstacle preventing the wide deployment of BFT systems in the cloud environment, as the legacy clients of Internet services can hardly integrate with this library.
- **Deploying and evaluating replicated services** in a cloud environment can be cumbersome, as managing such services is usually more complex and unfortunately not supported by most existing cloud services.

**Contributions** The main contributions of this thesis are listed below:

- A comprehensive study on how to overcome the **performance bottleneck of single-leader-based BFT protocols** with a parallel ordering framework. In particular, we focused on how to partition the service state and employ multiple leaders to exploit parallelism in a BFT system. In addition, a graph partitioning algorithm is applied to **dynamically reconfigure and update the state partitioning**, which ultimately improves performance.
- A detailed study on how to make **BFT systems transparent to clients** by using trusted hardware to implement a replacement of the client-side BFT library on the server side.
- **SAREK**, a parallel ordering framework with multiple leaders that partitions the service state to allow multiple BFT protocol instances to run independently at the same time, during both the agreement and execution stages. It predicts the partitions on which a request operates, and relies on the partitioning of the service state to allow concurrent processing of requests accessing different partitions of the state, according to a partition-specific schedule. When an inaccurate prediction occurs, SAREK is able

to handle it with a re-prediction. We have implemented a prototype of SAREK and conducted evaluations against a single-leader-based BFT system, and our results have shown that using SAREK can significantly improve performance. Moreover, our DYPART case studies have shown that improving the quality of state partitioning can ultimately lead to performance improvement of the parallel ordering framework. **DYPART** is a framework that maps the service state into multiple partitions and periodically reconfigures the partitions in the presence of dynamic usage patterns was presented. It relies on knowledge of the relationships between state objects for state partitioning and uses a high-performance graph partitioning algorithm to ensure that the resulting partitions can achieve both balanced workload and low synchronization across partitions. We implemented a prototype of DYPART based on SAREK and evaluated its performance in comparison.

- **TROXY**, a practical way to provide legacy clients with transparent access to replicated services by shifting traditional client-side BFT functions such as connection-handling and majority voting to the server side. It relies on a trusted subsystem to ensure its integrity and security. This is implemented based on Intel SGX technology and can only fail by crashing, so it can be trusted even in the face of malicious attacks. It also introduces a managed cache for read-heavy workloads to speed up read request processing, and is able to transparently switch to traditional request ordering in the event of write contention. A prototype of TROXY has been implemented that is fully transparent and secure to clients, and provides read-cache optimization without sacrificing linearizability. Evaluation results have shown that despite the overhead of using the trusted subsystem, TROXY outperforms a state-of-the-art BFT protocol with larger payloads, especially when network latency is applied as in a real Internet environment.
- **An automated deployment and evaluation framework** for BFT systems in a cloud environment that leverages the Metal-as-a-Service (MaaS) cloud infrastructure Ubuntu MAAS and the configuration management tool Ansible. This framework deploys physical machines with operating systems of the user's choice. By creating and executing configuration scripts with predefined tasks, this framework can automate the processes of installing essential runtime environments with different configurations, deploying and evaluating the selected implementations of BFT protocols, collecting evaluation results and creating plots etc. These operations are inserted as predefined tasks in the configuration scripts so that they can be automatically executed by the automation tool without requiring the user to issue manual commands during runtime. Our results have shown that this framework can be very efficient and effective for users to analyze their software configurations, and the results conducted with this framework are comparable to those of a manual installation.

## 7.2 Further Ideas

The following are some problems that can be addressed by modifying and extending the mechanisms presented in this thesis.

**Combine SAREK with TROXY** Our current implementations of SAREK and TROXY use different BFT protocols. In fact, the implementation of SAREK does not require any changes to the agreement protocol, so most BFT agreement protocols are compatible with it. As a result, it is actually possible to integrate the multi-leader-based agreement into TROXY by implementing the required components and features such as PREDICT(), the request scheduler and the re-prediction process on the server side, to achieve further performance improvement.

**Configuration Variance** In addition to the automated deployment and evaluation framework for BFT systems, we have previously conducted experiments with limited configuration variance, e.g., with two operating systems of different Linux distributions and three prototypes of BFT protocols: BFT-SMaRt, HYBSTER, and TROXY in the BFT system library. We assume that this framework can be extended in the next step to allow more diversity in terms of software configuration and integrated BFT systems. For example, if we increase the diversity of operating systems to include more distributions and versions, and integrate BFT systems written in programming languages other than Java, this will also result in different requirements for the runtime environment.

**BFT as a Service** Providing Byzantine fault tolerance guarantees for different applications is a major challenge for most existing cloud products, as it imposes additional management and resource requirements. We believe that the automated deployment and evaluation framework for BFT systems can be used as a foundation for building a BFT-as-a-Service cloud, called BFTaaS, that provides automated management of reliable applications with Byzantine fault tolerance. To achieve this, the current framework needs to be extended, especially in the client aspect. For example, a configuration manager component is needed to ensure that cloud offerings are configured according to user requirements and in a way that satisfies users. Exploration of common vulnerabilities of the system to reduce fault dependency can also be added to the service to allow users to optimize the configurations of their replicas.

# Bibliography


[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] P. Mell and T. Grance, "The nist definition of cloud computing," *NIST publication SP 800-145*, 2011.

[3] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 193–204.

[4] J. Dean, "Designs, lessons and advice from building large distributed systems," *Keynote from LADIS*, vol. 1, 2009.

[5] "Amazon s3 availability event," http://status.aws.amazon.com/s3-20080720.html, 2008, accessed: 2019-2-15.

[6] M. Osier, "Netflix blog-shipping delay recap," http://blog.netflix.com/2008/08/shipping-delay-recap.html, 2008, accessed: 2019-2-15.

[7] Pete, "Google app engine outage today," http://groups.google.com/group/google-appengine/browse_thread/thread/f7ce559b3b8b303b?pli=1, 2008, accessed: 2019-2-15.

[8] "Haproxy," http://www.haproxy.org/, 2011, accessed: 2019-2-9.

[9] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2003, pp. 235–248.

[10] P. Vijayan, N. Lakshmi, A. Nitin, S. Haryadi, C. Andrea, and H. Remzi, "Iron file systems," in *Proceedings of the twentieth ACM symposium on Operating systems principles (Brighton, United Kingdom*, 2005, pp. 206–220.

[11] I. Gashi, P. Popov, V. Stankovic, and L. Strigini, "On designing dependable services with diverse off-the-shelf sql servers," in *Architecting Dependable Systems II*. Springer, 2004, pp. 191–214.

[12] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden, "Tolerating byzantine faults in transaction processing systems using commit barrier scheduling," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 59–72.

[13] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[14] C. Cachin and J. A. Poritz, "Secure intrusion-tolerant replication on the internet," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 167–176.

[15] P. E. Veríssimo, N. F. Neves, and M. P. Correia, "Intrusion-tolerant architectures: Concepts and design," in *Architecting Dependable Systems*. Springer, 2003, pp. 3–36.

[16] P. E. Verissimo, N. F. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch, "Intrusion-tolerant middleware: The road to automatic security," *IEEE Security & Privacy*, vol. 4, no. 4, pp. 54–62, 2006.

[17] H. P. Reiser and R. Kapitza, "Hypervisor-based efficient proactive recovery," in *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*. IEEE, 2007, pp. 83–92.

[18] G. T. dos Santos Veronese, "Intrusion tolerance in large scale networks," Ph.D. dissertation, Universidade de Lisboa (Portugal), 2010.

[19] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin, "Bft: The time is now," in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. ACM, 2008, pp. 1–4.

[20] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 277–290.

[21] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: A hybrid quorum protocol for byzantine fault tolerance," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 177–190.

[22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM, 2007, pp. 45–58.

[23] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *International Conference on Dependable Systems and Networks, 2004*. IEEE, 2004, pp. 575–584.

[24] S. Sen, W. Lloyd, and M. J. Freedman, "Prophecy: Using history for high-throughput fault tolerance," in *NSDI*. USENIX Association, 2010, pp. 345–360.

[25] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*. IEEE, 2009, pp. 135–144.

[26] T. Distler and R. Kapitza, "Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 91–106.

[27] J. Behl, T. Distler, and R. Kapitza, "Consensus-oriented parallelization: How to earn your first million," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 173–184.

[28] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 253–267, 2003.

[29] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 59–74, 2005.

[30] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing*, no. 1, pp. 80–93, 2008.

[31] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin *et al.*, "All about eve: Execute-verify replication for multi-core servers," in *OSDI*, vol. 12. USENIX Association, 2012, pp. 237–250.

[32] R. Rodrigues, M. Castro, and B. Liskov, "Base: Using abstraction to improve fault tolerance," in *ACM SIGOPS Operating Systems Review*, ser. 35, no. 5. ACM, 2001, pp. 15–28.

[33] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 363–376.

[34] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "Cheapbft: resource-efficient byzantine fault tolerance," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 295–308.

[35] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: Sgx-based high performance bft," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 222–237.

[36] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, 2011.

[37] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *NSDI*, vol. 9. USENIX Association, 2009, pp. 153–168.

[38] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis *et al.*, "Zeno: Eventually consistent byzantine-fault tolerance," in *NSDI*, vol. 9. USENIX Association, 2009, pp. 169–184.

[39] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Resilient intrusion tolerance through proactive and reactive recovery," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*. IEEE, 2007, pp. 373–380.

[40] L. Lamport, "Generalized consensus and paxos," *Technical Report MSR-TR-2005-33, Microsoft Research*, 2005.

[41] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 358–372.

[42] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 368–377.

[43] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*. IEEE, 2014, pp. 57–66.

[44] C. E. Bezerra, F. Pedone, and R. Van Renesse, "Scalable state-machine replication," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2014, pp. 331–342.

[45] A. Bessani, H. P. Reiser, P. Sousa, I. Gashi, V. Stankovic, T. Distler, R. Kapitza, A. Daidone, and R. Obelheiro, "Forever: Fault/intrusion removal through evolution & recovery," in *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*. ACM, 2008, pp. 99–101.

[46] B.-G. Chun, P. Maniatis, and S. Shenker, "Diverse replication for single-machine byzantine-fault tolerance," in *USENIX Annual Technical Conference*, 2008, pp. 287–292.

[47] F. C. Gärtner, "Byzantine failures and security: Arbitrary is not (always) random," *INFORMATIK 2003-Mit Sicherheit Informatik, Schwerpunkt" Sicherheit-Schutz und Zuverlässigkeit"*, 2003.

[48] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[49] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga, "Depspace: a byzantine fault-tolerant coordination service," in *ACM SIGOPS Operating Systems Review*, ser. 42, no. 4. ACM, 2008, pp. 163–176.

[50] B. Li, W. Xu, M. Z. Abid, T. Distler, and R. Kapitza, "Sarek: Optimistic parallel ordering in byzantine fault tolerance," in *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 2016, pp. 77–88.

[51] B. Li, W. Xu, and R. Kapitza, "Dynamic state partitioning in parallelized byzantine fault tolerance," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 158–163.

[52] B. Li, N. Weichbrodt, J. Behl, P.-L. Aublin, T. Distler, and R. Kapitza, "Troxy: Transparent access to byzantine fault-tolerant systems," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 59–70.

[53] B. Li and R. Kapitza, "Bft-dep: automatic deployment of byzantine fault-tolerant services in paas cloud," in *Distributed Applications and Interoperable Systems*. Springer, 2016, pp. 109–114.

[54] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[55] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[56] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler, "Storyboard: Optimistic deterministic multithreading," in *HotDep*. USENIX Association, 2010.

[57] M. Castro and B. Liskov, "Proactive recovery in a byzantine-fault-tolerant system," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, vol. 4. USENIX Association, 2000.

[58] A. Avizienis and J.-C. Laprie, "Dependable computing: From concepts to design diversity," *Proceedings of the IEEE*, vol. 74, no. 5, pp. 629–638, 1986.

[59] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.

[60] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 189–204.

[61] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2807–2819, 2016.

[62] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc: Small trusted hardware for large distributed systems," in *NSDI*, vol. 9. USENIX Association, 2009, pp. 1–14.

[63] B. Vavala, N. Neves, and P. Steenkiste, "Securing passive replication through verification," in *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*. IEEE, 2015, pp. 176–181.

[64] N. Asokan, J.-E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "Mobile trusted computing," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1189–1206, 2014.

[65] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review*, vol. 37. ACM, 2003, pp. 193–206.

[66] GlobalPlatform, "Globalplatform technology tee system architecture version 1.1.0.10 (target v1.2)," http://globalplatform.org/wp-content/uploads/2018/09/GPD_TEE_SystemArch_v1.1.0.10-for-v1.2_PublicReview.pdf, 2018, accessed: 2019-2-15.

[67] A. Vasudevan, J. M. McCune, and J. Newsome, *Trustworthy execution on mobile devices*. Springer, 2014.

[68] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[69] C. Basile, Z. Kalbarczyk, and R. K. Iyer, "A preemptive deterministic scheduling algorithm for multithreaded replicas," in *DSN*. IEEE, 2003, pp. 149–158.

[70] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer, "Loose synchronization of multithreaded replicas," in *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*. IEEE, 2002, pp. 250–255.

[71] J. Pool, I. S. K. Wong, and D. Lie, "Relaxed determinism: Making redundant execution on multiprocessors practical," in *HotOS*. USENIX Association, 2007.

[72] Google, "Gmail," https://www.google.com/gmail/, 2004, accessed: 2019-2-15.

[73] "Dropbox," https://www.dropbox.com/, 2007, accessed: 2019-2-15.

[74] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 143–157.

[75] Google, "Google app engine," https://cloud.google.com/appengine/, 2008, accessed: 2019-2-15.

[76] "Red hat openshift," https://www.openshift.com/, 2011, accessed: 2019-2-15.

[77] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, ser. 37, no. 5. ACM, 2003, pp. 164–177.

[78] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[79] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor," in *USENIX Annual Technical Conference, General Track*, 2001, pp. 1–14.

[80] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 2–13, 2006.

[81] Amazon, "Amazon web service: On-demand cloud computing platforms," https://aws.amazon.com/, 2002, accessed: 2019-5-10.

[82] "Red hat openstack," https://www.openstack.org/, 2010, accessed: 2019-2-15.

[83] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Ebawa: Efficient byzantine agreement for wide-area networks," in *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*. IEEE, 2010, pp. 10–19.

[84] J. Sousa, A. Bessani, and M. Vukolic, "A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform," in *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2018, pp. 51–58.

[85] S. Meredith, "Whatsapp messenger service suffers major outage," https://www.cnbc.com/2017/11/03/whatsapp-down-messenger-service-suffers-major-outage.html, 2017, accessed: 2019-2-15.

[86] C. Mellor, "What did ovh learn from 24-hour outage? water and servers do not mix," https://www.theregister.com/2017/07/13/watercooling_leak_killed_vnx_array/, 2017, accessed: 2019-2-15.

[87] R. Ferraz, B. Gonçalves, J. Sequeira, M. Correia, N. F. Neves, and P. Veríssimo, "An intrusion-tolerant web server based on the distract architecture," in *In Proceedings of the Workshop on Dependable Distributed Data Management*. IEEE, 2004.

[88] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13. ACM, 2013.

[89] Intel, "Intel SGX," https://software.intel.com/en-us/sgx, 2017, accessed: 2017-9-8.

[90] G. Gebhart, "We're halfway to encrypting the entire web," https://goo.gl/em8elg, 2017, accessed: 2019-2-15.

[91] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[92] L. Lamport, "The Part-time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[93] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. ACM, 1988, pp. 8–17.

[94] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[95] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "Rbft: Redundant byzantine fault tolerance," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. IEEE, 2013, pp. 297–306.

[96] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 1–14, 2002.

[97] J. Cowling and B. Liskov, "Granola: low-overhead distributed transaction coordination," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, 2012, pp. 223–235.

[98] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 1–12.

[99] A. S. Tanenbaum and M. v. Steen, *Distributed systems: principles and paradigms*. Prentice Hall, 2007.

[100] D. E. Knuth, "Les misérables co-occurance network," https://people.sc.fsu.edu/jburkardt/datasets/sgb/jean.dat, 1994, accessed: 2018-2-8.

[101] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone, "Reconfiguring parallel state machine replication," in *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium on*. IEEE, 2017, pp. 104–113.

[102] O. M. Mendizabal, F. L. Dotti, and F. Pedone, "High performance recovery for parallel state machine replication," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 34–44.

[103] M. E. Newman, "Community detection and graph partitioning," *EPL (Europhysics Letters)*, vol. 103, no. 2, p. 28003, 2013.

[104] R. Glantz, H. Meyerhenke, and A. Noe, "Algorithms for mapping parallel processes onto grid and torus architectures," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 2015, pp. 236–243.

[105] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering*. Springer, 2016, pp. 117–158.

[106] P. Sanders and C. Schulz, "Think locally, act globally: Highly balanced graph partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, ser. LNCS, vol. 7933. Springer, 2013, pp. 164–175.

[107] C. Schulz, "Kahip: Karlsruhe high quality partitioning," http://algo2.iti.kit.edu/schulz/software_releases/kahipv2.00.pdf, 2013, accessed: 2018-2-8.

[108] "Kahip v2.0," https://github.com/schulzchristian/KaHIP/, 2013, accessed: 2018-2-8.

[109] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, *Graph partitioning and graph clustering*. American Mathematical Soc., 2013, vol. 588.

[110] DIMACS, "10th dimacs implementation challenge - graph partitioning and graph clustering," http://archive.dimacs.rutgers.edu/Workshops/Challenge10/, 2012, accessed: 2018-2-8.

[111] V. Viswanathan, "Load balancing web applications," http://www.oreillynet.com/pub/a/onjava/2001/09/26/load.html, 2001, accessed: 2019-6-10.

[112] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[113] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 143–158.

[114] G. Avoine, F. Gärtner, R. Guerraoui, and M. Vukolić, "Gracefully degrading fair exchange with security modules," in *European Dependable Computing Conference*. Springer, 2005, pp. 55–71.

[115] T. Distler, I. Popov, W. Schröder-Preikschat, H. P. Reiser, and R. Kapitza, "Spare: Replicas on hold," in *NDSS*, 2011.

[116] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-fault-tolerant middleware for web-service applications," in *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*. IEEE, 2005, pp. 131–140.

[117] W. Zhao, "Bft-ws: A byzantine fault tolerance framework for web services," in *EDOC Conference Workshop, 2007. EDOC'07. Eleventh International IEEE*. IEEE, 2007, pp. 89–96.

[118] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, 2014, pp. 305–320.

[119] Intel, "Intel SGX SDK," https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/get-started.html, accessed: 2017-9-8.

[120] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel sgx enclaves," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 201–213.

[121] S. Checkoway and H. Shacham, *Iago attacks: Why the system call api is a bad untrusted rpc interface*. ACM, 2013.

[122] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 440–457.

[123] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "Scone: Secure linux containers with intel sgx," in *OSDI*, 2016, pp. 689–703.

[124] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 238–253.

[125] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch, "Talos: Secure and transparent tls termination inside sgx enclaves," *Imperial College London, Tech. Rep*, vol. 5, 2017.

[126] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[127] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *26th USENIX security symposium (USENIX security 17)*, 2017, pp. 557–574.

[128] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in *NDSS*, 2017.

[129] A. Mahimkar, J. Dange, V. Shmatikov, H. M. Vin, and Y. Zhang, "dfence: Transparent network-based denial of service mitigation," in *NSDI*, vol. 7, 2007, pp. 327–340.

[130] Apache, "Apache jmeter," http://jmeter.apache.org/, accessed: 2020-6-8.

[131] Eclipse, "Embedded jetty v.9.4," https://github.com/eclipse/jetty.project, accessed: 2020-6-8.

[132] "Docker: Build, ship, and run any app, anywhere," https://www.docker.com/, 2013, accessed: 2019-7-15.

[133] "Docker documentation," https://docs.docker.com/engine/docker-overview/, 2018, accessed: 2019-7-15.

[134] "Docker swarm," https://docs.docker.com/engine/swarm/, 2018, accessed: 2019-7-15.

[135] Google, "Kubernetes: Production-grade container orchestration," https://kubernetes.io/, 2014, accessed: 2019-7-15.

[136] "Red hat software," https://www.redhat.com/en, 1993, accessed: 2019-7-15.

[137] "Red hat openshift online for public cloud application hosting," https://manage.openshift.com/, 2018, accessed: 2019-7-15.

[138] "Red hat openshift container platform," https://www.openshift.com/products/container-platform/, 2018, accessed: 2019-7-15.

[139] Ubuntu, "Maas: Metal as a service," https://maas.io/, 2012, accessed: 2019-7-15.

[140] "Red hat ansible: Simple it automation," https://www.ansible.com/, 2012, accessed: 2019-7-15.

[141] Chef, "Devops dashboard for complete operational visibility into the coded enterprise," https://www.chef.io/, 2009, accessed: 2019-7-15.

[142] Puppet, "Make infrastructure actionable, scalable and intelligent," https://puppet.com/, 2005, accessed: 2019-7-15.

[143] Amazon, "Ec2: Amazon elastic compute cloud," https://aws.amazon.com/ec2/, 2006, accessed: 2019-7-15.

[144] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," *Acm transactions on storage (tos)*, vol. 9, no. 4, pp. 1–33, 2013.

[145] V. V. Cogo, A. Nogueira, J. Sousa, M. Pasin, H. P. Reiser, and A. Bessani, "Fitch: supporting adaptive replicated services in the cloud," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2013, pp. 15–28.

[146] P. Garraghan, P. Townend, J. Xu, X. Yang, and P. Sui, "Using byzantine fault-tolerance to improve dependability in federated cloud computing," *International Journal of Software and Informatics*, vol. 7, no. 2, pp. 221–237, 2013.

[147] P. Verissimo, A. Bessani, and M. Pasin, "The tclouds architecture: Open and resilient cloud-of-clouds computing," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. IEEE, 2012, pp. 1–6.

[148] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "Bft protocols under fire," in *NSDI*, vol. 8. USENIX Association, 2008, pp. 189–204.

[149] "ns-2 network simulator," http://www.isi.edu/nsnam/ns/, 2000, accessed: 2019-7-15.

[150] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru, "Turret: A platform for automated attack finding in unmodified distributed system implementations," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 660–669.

[151] "ns-3 network simulator," http://www.nsnam.org/, 2008, accessed: 2019-7-15.

[152] D. Gupta, L. Perronne, and S. Bouchenak, "Bft-bench: Towards a practical evaluation of robustness and effectiveness of bft protocols," in *Distributed Applications and Interoperable Systems*. Springer, 2016, pp. 115–128.

[153] M. Garcia, A. Bessani, and N. Neves, "Lazarus: Automatic management of diversity in bft systems," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 241–254.

[154] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 355–362.